



FME® Functions and Factories


**(Reference for Functions and Factories supported
by FME)**



SAFE SOFTWARE

E-mail: info@safe.com • Web: www.safe.com

Change Bars

 From time to time you will see red change bars on the left-hand side of a chapter. This means that the text has been updated since our last release of FME.

Safe Software Inc. makes no warranty either expressed or implied, including, but not limited to, any implied warranties of merchantability or fitness for a particular purpose regarding these materials, and makes such materials available solely on an “as-is” basis.

In no event shall Safe Software Inc. be liable to anyone for special, collateral, incidental, or consequential damages in connection with or arising out of purchase or use of these materials. The sole and exclusive liability of Safe Software Inc., regardless of the form or action, shall not exceed the purchase price of the materials described herein.

This manual describes the functionality and use of the Feature Manipulation Engine at the time of publication. The software described herein and the descriptions themselves, are subject to change without notice.

Copyright

Copyright © 2000–2009 Safe Software Inc. All rights are reserved.

Revisions

Every effort has been made to ensure the accuracy of this document. Safe Software Inc. regrets any errors and omissions that may occur and would appreciate being informed of any errors found. Safe Software Inc. will correct any such errors and omissions in a subsequent version, as feasible. Please contact us at:

Safe Software Inc.

Fax: 604-501-9965

www.safe.com

Safe Software Inc. assumes no responsibility for any errors in this document or their consequences, and reserves the right to make improvements and changes to this document without notice.

Trademarks

FME is a registered trademark of Safe Software Inc. Other companies and products mentioned herein may be trademarks or registered trademarks of their respective trademark owners.

Document Information

Document Name: FME Functions and Factories

Version: FME 2009

December 2008

CONTENTS

About This Help File xxv

Intended Audience	xxv
Navigating the Functions and Factories Help	xxv
For More Information	xxv

FME Functions 1

@Abort	3
Configuration	3
Description	3
Inverse Operation	3
Example	4
@AddRasterBand	6
Configuration	6
Description	6
Inverse Operation	7
Example	7
@AddVertices	9
Configuration	9
Description	9
Inverse	10
Example	10
@Affine	11
Configuration	11
Description	11
Inverse Operation	12
Example	12
@Angle	14
Configuration	14
Description	14
Inverse Operation	15
Example	15
@ApplyRasterRotation	16
Configuration	16
Description	16
Inverse Operation	17
Example	17
@Arc	18
Configuration	22
Description	22
Inverse Operation	22
Example	23
@Area	24
Configuration	24
Description	24

Inverse Operation	24
Example	24
@Bounds	25
Configuration	25
Description.	25
Inverse Operation	26
Example	26
@Buffer	28
Configuration	28
Description.	28
Inverse Operation	29
Example	30
@Buffer2	32
Configuration	32
Description.	32
Inverse Operation	34
Example	34
@CheckpointRaster	36
Configuration	36
Description.	36
Inverse Operation	36
Example	36
@Circularity	37
Configuration	37
Description.	37
Inverse Operation	37
Example	37
@Close	38
Configuration	38
Description.	38
Inverse Operation	38
Example	39
@Concatenate	40
Configuration	40
Description.	40
Inverse Operation	41
Example 1	41
Example 2	42
Example 3	42
Example 4	42
@ConvertBase	43
Configuration	43
Description.	43
Inverse	44
Example	44
@ConvertToArc	45
Configuration	46
Description.	46
Inverse	47
Example 1	47
Example 2	47
Example 3	48

Example 4	48
@ConvertToLine	49
Configuration	49
Description.	49
Inverse Operation.	50
Example	50
@ConvertToPoint	51
Configuration	51
Description.	51
Inverse Operation.	51
Example	51
@ConvexHull	52
Configuration	52
Description.	52
Inverse Operation.	52
Example	52
@Coordinate	53
Description.	54
Inverse	55
Example	55
@CoordSys	57
Configuration	57
Description.	58
Inverse Operation.	58
Example	58
@CopyAttributes	59
Configuration	59
Description.	59
Inverse Operation.	59
Example 1	60
Example 2	60
@Count	61
Configuration	61
Description.	62
Inverse Operation.	62
Example	62
@CRC	64
Configuration	64
Description.	64
Inverse Operation.	64
Example 1	65
Example 2	65
Example 3	65
@CSG	66
Configuration	66
Description.	66
Inverse Operation.	66
@Dimension	67
Configuration	67
Description.	67
Inverse Operation.	67
Example	67

@Evaluate	68
Configuration	68
Description.	68
Operands.	69
Operators	69
Math Functions.	71
Types, Overflows, Precision	71
Inverse Operation.	72
Example	72
@FeatureType	73
Configuration	73
Description.	73
Inverse Operation.	74
Example	74
@File	75
Configuration	76
Description.	76
Inverse Operation.	76
Example	77
@Filenamepart	78
Configuration	78
Description.	78
Inverse Operation.	78
Example	79
@FitDGN	80
Configuration	80
Description.	80
Inverse Operation.	81
Example	81
@Force2D	82
Configuration	82
Description.	82
Inverse Operation.	82
Example	82
@Generalize	83
Configuration	85
Description.	85
Inverse Operation.	87
Example	87
@GeneratePoint	88
Configuration	88
Description.	88
Inverse Operation.	88
Example	89
@GenerateRasterPalettes	90
Configuration	90
Description.	90
Inverse Operation.	91
Example	92
@Geometry	94
Configuration	95
Description.	95

Inverse Operation	97
@GeometryTraits	98
Description.	99
Inverse Operation.	100
Example	100
@GeometryType	102
Description.	104
Inverse Operation.	106
Example	106
@GeoreferenceRaster	108
Configuration	109
Description.	109
Inverse Operation.	109
Example	109
@GlobalVariable	111
Configuration	111
Description.	111
Inverse Operation.	112
Example	112
@GOID	113
Configuration	113
Description.	113
Inverse Operation	114
Formal Specification	114
Encoding	116
Example 1	116
Example 2	116
References.	117
@GridSnapper	118
Configuration	118
Description.	118
Inverse Operation.	119
Example	119
@Http	120
Description.	122
Inverse Operation.	123
Example	123
@Interpolate	124
Configuration	124
Description.	124
Inverse Operation.	124
Example	124
@KeepAttributes	126
Configuration	126
Description.	126
RE Construction	127
Inverse Operation.	128
Example	128
@Length	129
Configuration	129
Description.	130
Inverse Operation.	130

Example	130
@Log	131
Configuration	131
Description.	132
Inverse Operation.	132
Example	132
@Lookup	133
Configuration	133
Description.	134
Inverse Operation.	134
Example 1	135
Example 2	136
@MergeLists	137
Configuration	137
Description.	137
Inverse Operation.	137
Example	137
@MGRS	140
Configuration	141
Description.	141
Inverse Operation.	142
Example	142
@MinimumSpanningCircle	144
Configuration	144
Description.	144
Inverse Operation.	144
Example	144
@NumCoords	145
Configuration	145
Description.	145
Inverse Operation.	145
Example	145
@NumElements	146
Configuration	146
Description.	146
Inverse Operation.	146
Example	147
@NumHoles	148
Configuration	148
Description.	148
Inverse	148
Example	148
@Offset	149
Configuration	149
Description.	149
Inverse Operation.	149
Example	149
@OGCGeometry	150
Configuration	150
Description.	150
Example	152
@OrderRasterBands	154

Configuration	154
Description.	154
Inverse Operation.	154
Example	154
@Orient	156
Configuration	156
Description.	156
Inverse	157
Example:	157
@ProcessRasterTiles	158
Configuration	158
Description.	159
Inverse Operation.	159
Example 1	159
@Python	161
Configuration	161
Description.	161
Example	163
@RasterCellOrigin	166
Configuration	166
Description.	166
Inverse Operation.	166
Example	166
@RasterGCP	168
Configuration	168
Description.	168
Inverse Operation.	169
Example 1	169
Example 2	169
@RasterGeometry	170
Configuration	170
Description.	170
Inverse Operation.	171
Example	171
@RasterNodata	172
Configuration	172
Description.	172
Inverse Operation.	173
Example 1	173
Example 2	173
Example 3	173
@RasterProperties	176
Configuration	176
Description.	176
Inverse Operation.	177
Example 1	177
@Reformat	178
Configuration	178
Description.	179
Inverse Operation.	179
Example	180
@RemoveRasterBands	182

Configuration	182
Description.	182
Inverse Operation.	182
Example	182
@RemoveRasterPalettes	184
Configuration	184
Description.	184
Inverse Operation.	184
Example	184
@ReinterpretRaster	186
Configuration	187
Description.	188
Inverse Operation.	189
Example	189
@ReplaceRasterCellValues	192
Configuration	193
Description.	193
Inverse Operation.	193
Example	193
@Relate	195
Description.	195
Inverse Operation.	197
Configuration	198
Example 1	205
Example 2	207
Example 3	208
Example 4	214
@RemoveAttributes	216
Description.	216
Inverse Operation.	217
Example	217
@RemoveGeometry	218
Configuration	218
Description.	218
Inverse Operation.	218
Example	218
@RenameAttributes	219
Configuration	219
Description.	220
Inverse Operation.	220
Example	220
@ReplaceCoordinates	221
Configuration	221
Description.	221
Inverse Operation.	221
Example	221
@Reproject	223
Description.	224
Inverse Operation.	226
Example	226
@ReprojectAngle	227
Description.	227

Inverse Operation	227
Example	228
@ReprojectLength	229
Description.	229
Inverse Operation.	229
Example	230
@ResampleRaster	232
Configuration	232
Description.	233
Inverse Operation.	233
Example	233
@ResolveRasterPalettes	234
Description.	234
Inverse Operation.	234
Example	234
@Rotate2D	235
Configuration	235
Description.	235
Inverse Operation.	235
Example	236
@RoundOffCoords	237
Configuration	238
Description.	238
Inverse Operation.	238
Example	238
@Scale	239
Configuration	239
Description.	239
Inverse Operation.	240
Example	240
@SDEsql	241
Configuration	241
Connection Parameters	241
Description.	244
Assumptions	244
Inverse Operation.	244
Example	244
@SearchList	246
Configuration	247
Description.	247
Inverse Operation.	247
Example	248
@SecondOrderConformal	249
Configuration	249
Description.	249
Inverse Operation.	250
Example	250
@SelectRaster	252
Configuration	252
Description.	252
Inverse Operation.	252
Example 1	253

Example 2	253
Example 3	253
Example 4	253
@Snip	255
Configuration	256
Description.	256
Features	256
@Split	258
Configuration	259
Description.	259
Inverse Operation	260
Example	261
@SQL	263
Configuration	263
Description.	264
Error Handling	265
Inverse Operation	265
Example 1	265
Example 2	266
@SubsetRaster	268
Configuration	268
Description.	269
Inverse Operation	269
Example	269
@SupplyAttributes	270
Configuration	270
Description.	270
Inverse Operation	271
Example	271
@System	272
Configuration	272
Description.	272
Inverse Operation	272
Example	273
@TCL	274
Configuration	274
Description.	275
Global Variables	275
FME_Attributes Array	276
FME_FeatureType Variable	276
FME_CoordSys Variable	277
FME_AttrEncoding Variable	277
FME_AttrNameEncoding Variable	277
FME_Coordinates Function	278
FME_LogMessage Function	279
FME_Execute function	279
Inverse Operation	280
Example	280
@Tcl2	283
Configuration	283
Description.	284
FME_AttrEncoding Variable	285

FME_AttributeExists function	285
FME_AttrNameEncoding Variable	286
FME_AttributeNames function	286
FME_CoordSys Variable	286
FME_Coordinates Function	287
FME_CopyAttribute function	287
FME_Execute function	288
FME_FeatureType Variable	288
FME_GetAttribute function	288
FME_LogMessage Function	288
FME_RenameAttribute function	289
FME_SetAttribute function	289
FME_TempFilename function	289
FME_UnsetAttributes function	290
Global Variables	290
Inverse Operation	291
Example	291
@Timestamp	294
Configuration	294
Description	294
Inverse Operation	295
Example	295
@Transform	297
Configuration	297
Description	297
Inverse Operation	298
Example	298
@UUID	299
Configuration	299
Description	299
Example	299
@Value	300
Configuration	300
Description	300
Inverse Operation	300
Example	300
@XValue	302
Configuration	302
Description	302
Inverse Operation	303
Example	303
Example: Setting Only the X Value on a Text Feature	305
@YValue	306
Configuration	306
Description	306
Inverse Operation	306
Example	307
Example: Setting Only the Y Value on a Text Feature	307
@ZValue	308
Configuration	308
Description	308
Inverse Operation	308

Example	309
---------------	-----

FME Factories	311
----------------------	------------

AggregateFactory	313
Syntax.	313
Overview	313
Assumptions	315
Output Tags	315
Example	315
ArcFactory	317
Syntax.	317
Overview	317
Assumptions	318
Output Tags	319
Example	319
BoundingBoxFactory	320
Syntax.	320
Overview	320
Assumptions	321
Output Tags	321
Example	321
AttributeFactory	323
Syntax.	323
Overview	323
Assumptions	323
Output Tags	324
Example	324
BranchingFactory	325
Syntax.	325
Overview	325
Assumptions	326
Output Tags	326
ChoppingFactory	327
Syntax.	327
Overview	327
Assumptions	328
Output Tags	328
Example	328
ClippingFactory	330
Syntax.	330
Overview	330
Example	333
CloseFactory	334
Syntax.	334
Overview	334
Assumptions	334
Input Tags	335
Output Tags	335
Example	336
CommonSegmentFactory	337
Syntax.	337

Overview	337
Assumptions	337
Input Tags	338
Output Tags	338
Example	338
ConnectionFactory	339
Syntax.	339
Overview	339
Assumptions	341
Output Tags	341
Example	342
ConvexHullFactory	343
Syntax.	343
Overview	343
Assumptions	344
Output Tags	344
Example	344
CorrelationFactory	345
Syntax.	345
Overview	345
Assumptions	347
Output Tags	348
Example 1	348
Example 2	349
CreationFactory	351
Syntax.	351
Overview	351
Assumptions	352
Output Tags	352
Example	352
CSGFactory	354
Syntax.	354
Overview	354
Output Tags	354
Examples.	354
DeaggregateFactory	356
Syntax.	356
Overview	356
Assumptions	357
Output Tags	358
Example	358
DEMDistanceFactory	360
Syntax.	360
Overview	360
DEM_DIST.	360
Assumptions	360
Output Tag	361
DonutFactory	362
Syntax.	362
Overview	362
Assumptions	364
Output Tags	364

Example	365
DonutHoleFactory	366
Syntax.	366
Overview	366
Assumptions	366
Output Tags	366
Example	367
ElementFactory	368
Syntax.	368
Overview	368
CLASSIC Mode	369
LEAN Mode.	369
LEAN_AND_MEAN Mode.	369
Assumptions	370
Output Tags	370
Example	370
ExtensionFactory	373
Syntax.	373
Overview	373
Assumptions	374
Output Tags	374
Example	375
GeoRSSFactory	377
Syntax.	377
Overview	377
Assumptions	379
Output Tags	379
Example	380
GML2Factory	381
Syntax.	381
Overview	381
Assumptions	382
Output Tags	382
Example	383
GridToVectorFactory	384
Syntax.	384
Overview	384
Assumptions	384
Output Tags	385
IntersectionFactory	386
Syntax.	386
Overview	386
Assumptions	389
Output Tags	389
Example 1	389
Example 2	389
JSONQueryFactory	391
Syntax.	391
Overview	391
JSON Queries	391
JSON Structure Expressions.	391
Extract Mode	392

Output Tags	393
Examples	393
LabelFactory	395
Syntax	395
Overview	395
Assumptions	397
Output Tags	397
Example	397
ListFactory	398
Syntax	398
Overview	398
Assumptions	398
Output Tags	399
Example	399
LLOutlineAndCentroidFactory	401
Syntax	401
Overview	401
Output Tags	402
Example	402
MatchingFactory	403
Syntax	403
Overview	403
Assumptions	403
Clauses	404
Output Tags	407
Example	407
MRFCleanFactory	410
Syntax	410
Overview	410
Module Sequence	410
General Processing Tips	412
Know your data	412
Start small	412
Mix it up	412
MRFCleanFactory Modules	412
TOLERANCE	412
SIMPLIFY	412
SHORT_ELEMENT	413
EXTEND	413
INTERSECT	416
DUPLICATE_REMOVE	416
JOIN	416
CONFLATE	416
DANGLER	416
OBJECT_CLEAN	416
Sample Results	417
Assumptions	420
Output Tags	420
NeighborColorSetterFactory	422
Syntax	422
Overview	422
ALGORITHM	422

COLOR_ID_ATTR	422
SET_COLORS	422
AREA_ID_ATTR and NEIGHBOR_IDS_ATTR	423
Assumptions	423
Output Tags	423
OracleQueryFactory	424
Syntax.	424
Overview	424
OracleQueryFactory Processing.	425
Assumptions	426
Clauses	426
Specifying Spatial Relationships	427
Output Tags	429
Example	429
OverlayFactory	430
Syntax.	430
Overview	430
Mode 1: Polygon Overlay	431
Mode 2: Point on Polygon Overlay.	431
Mode 3: Line on Polygon Overlay	431
Mode 4: Point on Line Overlay	432
Mode 5: Point Overlay	432
Mode 6: Line Overlay	433
Assumptions	433
Input Tags	433
Output Tags	434
Example 1	434
Example 2	434
Example 3	435
PIPComponentsFactory	436
Syntax.	436
Overview	436
Assumptions	436
Output Tags	436
Example	437
PolygonDissolveFactory	438
Syntax.	438
Assumptions	439
Output Tags	439
Example 1	441
Example 2	441
PolygonFactory	443
Syntax.	443
Overview	443
Assumptions	445
Output Tags	445
Example	445
ProximityFactory	446
Syntax.	446
Overview	446
Assumptions	448
Output Tags	449

Example	449
PythonFactory	451
Syntax	451
Description	451
Assumptions	451
Clauses	451
Output Tags	452
Python Implementation Objects	452
Python Class Implementation	452
Python Function Implementation	453
Python Object Implementation	453
Example	454
Example 1: Factory with Python Function Implementation	454
Example 2: Factory with Python Class Implementation	454
Example 3: Factory with Object Implementation	455
QueryFactory	456
Syntax	456
Overview	456
Reader Specification	458
Consecutive Queries	459
Example	460
RasterClippingFactory	462
Syntax	462
Overview	462
Assumptions	463
Output Tags	463
Examples	464
RasterCreationFactory	466
Syntax	466
Overview	466
Assumptions	468
Output Tags	469
Example	469
RasterEvaluationFactory	472
Syntax	472
Overview	472
Operands	473
Operators	474
Functions	475
Assumptions	476
Output Tags	476
Examples	476
RasterMergerFactory	478
Syntax	478
Overview	478
Assumptions	479
Output Tags	479
Example	479
RasterMosaicFactory	480
Syntax	480
Overview	480
Output Tags	482

Example	482
RasterPyramidFactory	484
Syntax.	484
Overview	484
Assumptions	485
Output Tags	485
Examples	485
RasterSplitterFactory	488
Syntax.	488
Overview	488
Assumptions	489
Output Tags	489
Example	489
RasterSubsetFactory	490
Syntax.	490
Overview	490
Assumptions	491
Output Tags	491
Examples	491
RasterToVectorFactory	492
Syntax.	492
Overview	492
Assumptions	493
Output Tags	493
Example	493
RecorderFactory	494
Syntax.	494
Overview	494
Recording	495
Playing Back.	496
Assumptions	496
Output Tags	496
Example 1	496
Example 2	497
ReferenceFactory	499
Syntax.	499
Overview	499
Assumptions	502
Input Tags	502
Clauses	502
Output Tags	504
Example 1	506
Example 2	507
ReportFactory	508
Syntax.	508
Overview	508
Assumptions	508
Clauses	508
Output Tags	509
Example	510
SamplingFactory	511
Syntax.	511

Overview	511
Assumptions	512
Output Tags	512
Example	512
SDE30QueryFactory	513
Syntax.	513
Overview	513
SDE30QueryFactory Processing	515
Assumptions	515
Connecting to SDE	515
Regular Connection	516
Direct Connection	516
Clauses	519
Output Tags	527
Using Versioning with the SDE30 Reader, Writer, and QueryFactory	527
Example	527
SectorFactory	530
Syntax.	530
Overview	530
Assumptions	532
Clauses	532
Output Tags	533
Example	534
ServerJobSubmissionFactory	535
Syntax.	535
Overview	535
Assumptions	535
Clauses	536
Output Tags	536
ServerJobWaitingFactory	539
Syntax.	539
Overview	539
Assumptions	539
Clauses	540
Output Tags	540
SmallworldGeometryFactory	541
Syntax.	541
Overview	541
Assumptions	542
Output Tags	542
Output Feature Attribution	543
Deaggregation of Chains and Areas.	544
Example	544
SnappingFactory	545
Syntax.	545
Overview	545
Assumptions	546
Output Tags	546
Example	548
SortFactory	549
Syntax.	549
Overview	549

Assumptions	549
Output Tags	550
Example	550
SpatialFilterFactory	552
Syntax.	552
Overview	552
Assumptions	553
Input Tags	553
Clauses	553
Spatial Predicate Background	554
BOUNDARY, INTERIOR AND EXTERIOR	555
Definitions of PREDICATE	556
Output Tags	558
Example 1	559
Example 2	559
Example 3	560
SpatialRelationshipFactory	562
Syntax.	562
Overview	562
Assumptions	563
Input Tags	563
Clauses	563
Output Tags	565
SpikeRemoverFactory	568
Syntax.	568
Overview	568
Assumptions	569
Output Tags	569
Examples	569
SurfaceModelFactory	570
Syntax.	570
Overview	570
Assumptions	572
Input Tags	572
Output Tags	572
Example	574
TeeFactory	575
Syntax.	575
Overview	575
Assumptions	575
Output Tags	575
Example	576
TestFactory	577
Syntax.	577
Overview	577
Assumptions	578
Output Tags	578
Example	578
TextStrokerFactory	580
Syntax.	580
Overview	580
Assumptions	580

Clauses	581
Output Tags	582
Example	582
TilingFactory	584
Syntax.	584
Overview	584
Assumptions	585
Output Tags	585
Example	585
TopologyFactory	586
Syntax.	586
Overview	586
Assumptions	591
Output Tags	591
Example	593
TriangulationFactory	594
Syntax.	594
Overview	594
Assumptions	594
Output Tags	594
.	594
Geometry Type of Output Geometry Container	595
Examples.	596
VectorOnRasterOverlayFactory	598
Syntax.	598
Overview	598
POINT_ON_RASTER	598
Assumptions	598
Output Tags	599
VectorToRasterFactory	600
Syntax.	600
Overview	600
Assumptions	601
Input Tags	602
Output Tags	602
Example	602
VirtualEarthTileFactory	604
Syntax.	604
Overview	604
Assumptions	605
Output Tags	605
Examples.	605
WarpFactory	606
Syntax.	606
Overview	606
Output Tags	608
Control Vector Creation	608
Example	609
XFMapFactory	611
Syntax.	611
Overview	611
Output Tags	612

Examples	612
XPathFactory	613
Syntax	613
Overview	613
XPath Expressions	614
Extract Mode	615
Explode Mode	616
Output Tags	616
Examples	616
XQueryFactory	619
Syntax	619
Overview	619
Clauses	620
Extract Mode	622
Explode Mode	623
Output Tags	623
Examples	623
XSLTFactory	628
Syntax	628
Overview	628
Assumptions	629
Clauses	629
Output Tags	630
Examples	630
 About Workbench Transformers	 633
Transformer Help	634
Transformer Cross-References	634
 Quick Reference	 635
Functions	635
Factories	638
List of Directives	640
List of Environment Variables	642
 Syntax of Tcl Regular Expressions	 643
Different Flavors of REs	643
Regular Expression Syntax	643

About This Help File

Intended Audience

This manual is intended for a technical audience involved in configuring the FME for customized translation. FME's functions and factories will be of particular interest to data managers, software professionals, and advanced Geographic Information Systems (GIS) operators who wish to maximize the value of their data by processing it using FME facilities.

In most cases, however, Transformers (which are the FME Workbench equivalent of functions and factories) will be sufficient when performing any data transformations. Workbench transformers are documented in the Workbench on-line help files, and this manual can be used as a supplemental reference.

Navigating the Functions and Factories Help

This manual is divided into three main chapters:

- FME's functions are used in FME mapping files and operate on one feature at a time, producing a value or changing the attributes or geometry of the feature. The ***FME Functions*** section contains a detailed description of each FME function, and is arranged alphabetically by function name.
- FME's factories are used in FME mapping files and operate on many features at a time, producing many features as output. The ***FME Factories*** section contains a detailed description of each FME factory, and is arranged alphabetically by factory name.
- Workbench transformers are built upon FME functions and factories. The ***Workbench Transformer*** section contains a cross-referenced list of transformers and functions/factories. For details on each transformer, press the F1 key after inserting a transformer into your workspace.

For More Information

For more information about FME, refer to www.safe.com.

FME Functions

FME Functions operate on individual FME features, calculating values, or modifying attributes or geometry. A wide range of functions is available for use in FME mapping files. This chapter provides detailed information on each FME function, and describes the parameters it takes, the operation it performs, and the value it returns.

@Abort

@Abort ([<message>])

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<message>	Any String	The string printed when the abort function is invoked. This message can be used to simply identify that the abort function was called, or to print a message that describes the reason the translation was aborted.	Yes

Configuration

The @Abort function does not accept configuration lines.

Description

This function provides a mechanism for selectively aborting a data translation. @Abort is typically used to abort a data translation when erroneous data is encountered. This is useful when using the FME for quality assurance purposes.

The optional `message` parameter can be used to print the reason the data translation is aborted. For example, if there are several abort functions used throughout a mapping file, then the `message` should clearly state the reason the translation was aborted and identify the particular abort command that was triggered. If no `message` is specified, then the translation will be aborted and no message will be printed.

Tip

@Abort is most often used on a factory `OUTPUT` clause that should never be invoked unless the input data was bad. Using @Abort ensures the translation will fail and there is no chance the bad data will go unnoticed.

The @Abort function also logs the feature upon which it was invoked.

Inverse Operation

This function works identically in both forward and reverse directions.

Example

In the example below, @Abort is used with the ReferenceFactory to terminate the translation when several different erroneous conditions are encountered.

```

FACTORY_DEF Shape ReferenceFactory                                \
  INPUT REFERENCEE FEATURE_TYPE boundary                        \
                                                                    \
  INPUT REFERENCER FEATURE_TYPE forestCover                     \
                                                                    \
  REFERENCEE_FIELDS lineID                                       \
  REFERENCER_FIELDS arcs{}                                       \
  REFERENCE_INFO GEOMETRY                                        \
  OUTPUT COMPLETE FEATURE_TYPE *                               \
  OUTPUT INCOMPLETE FEATURE_TYPE *                             \
    @Abort("Incomplete feature found in ReferenceFactory") \
  OUTPUT UNREFERENCED FEATURE_TYPE *                           \
    @Abort("Unreferenced boundary arc found")                 \
  OUTPUT NO_REFERENCEE_FIELD FEATURE_TYPE *                    \
    @Abort("Boundary with no reference field")                 \
  OUTPUT DUPLICATE_REFERENCEE FEATURE_TYPE *                   \
    @Abort("Two Boundary arcs found with the same id")

```

In the second example, @Abort is used in a similar fashion with the PolygonFactory to terminate the translation when any extraneous linework, that does not play a role in the formation of any polygons, is encountered. As part of the OUTPUT LINE clause, it is activated as soon as any unclosed linework is output by the factory. This can be used to flag possible overshoots, or undershoots, in what should otherwise be properly noded and closed input linework.

```

FACTORY_DEF IGDS PolygonFactory                                \
  END_NODDED                                                    \
  INPUT FEATURE_TYPE 33 igds_type igds_line                    \
  OUTPUT POLYGON FEATURE_TYPE FormedPolygon                    \
  OUTPUT LINE FEATURE_TYPE ExtraLines                          \
    @Abort("Extra Linework encountered by PolygonFactory")

```


@AddRasterBand

```
@AddRasterBand(<interpretation>, <cell value>, <nodata value>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<interpretation>	(RED8 RED16 GREEN8 GREEN16 BLUE8 BLUE16 ALPHA8 ALPHA16 GRAY8 GRAY16 INT8 INT16 INT32 INT64 UINT8 UINT16 UINT32 UINT64 REAL32 REAL64)	The desired interpretation for the added band, including data type and bit depth.	No
<cell value>	Numeric	The value that will be used for all cells in the added band.	No
<nodata value>	Numeric	The nodata value for the band. If left blank, the band will not be given a nodata value.	Yes

Configuration

This function does not accept configuration lines.

Description

The @AddRasterBand function is used to add a new band to a raster. The added band will have the same value in all cells. Furthermore, the added band will have the same raster-level properties as other bands in the raster (i.e. number of rows/columns, cell spacing, cell origin, etc.).

For example, if you had a raster with RGB bands, but you wanted to write the raster as RGBA, you could use this function to add an Alpha band to the raster.

The <interpretation> parameter specifies the desired interpretation for the added band.

The <cell value> parameter sets the value that will be used for all cells in the added band.

The <nodata value> parameter sets the band's nodata value. If a nodata value is not desired, this parameter may be left blank.

Input features must contain raster geometries only.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a typical use of the @AddRasterBand function. In this example, the `SamplingFactory` is used to pass through every feature, while the @AddRasterBand function adds an alpha band which has a value of 255 in every cell, and a nodata value of 0.

```
FACTORY_DEF * SamplingFactory          \  
    SAMPLE_RATE 1                      \  
    INPUT FEATURE_TYPE *              \  
        @AddRasterBand(ALPHA8, 255, 0) \  
    OUTPUT FEATURE_TYPE *
```


@AddVertices

@AddVertices ((x|y|xy) , <interval>)

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<axis>	(x y xy)	The first parameter controls the axis used to determine when vertices are added. If xy is specified, then vertices will be added on the basis of the 2D length of segments. The x and y axes may be specified to generate tick marks along each of them at multiples of the interval.	No
<interval>	Real Number	The second parameter specifies the interval at which vertices are added. It is used differently depending on the setting of the <axis> parameter. If the <axis> is x or y, then the new vertices will be inserted into the feature along the specified axis at each multiple of <interval>. If the <axis> is xy, then new vertices will be inserted so no single line segment in the feature is longer than the <interval>. The interval is measured in the feature's ground units.	No

Configuration

The @AddVertices function does not accept configuration lines.

Description

This function adds vertices to features. It interpolates new coordinates at some specified interval. The interval may be along only one of the two primary axes, or it may be along the length of the line segments.

This function is often used to densify the vertices of a feature to prepare it for reprojection. By adding vertices to long linear segments, the feature better represents the original in a different coordinate system.

When used with the x or y option, the function may also be used to add tick marks to features along an axis at the specified <interval>.

If the FME_GEOMETRY_HANDLING directive is set to “yes”, arc segments in a path are preserved and not densified. Otherwise, paths are stroked into line before adding the new vertices.

Inverse

This function has no inverse.

Example

In the example below, new vertices are added at even multiples of 10 along the x axis only.

```
FACTORY_DEF SHAPE SamplingFactory \
INPUT FEATURE_TYPE * \
    @AddVertices(x,10)
```

If a feature was input with these coordinates:

```
(7,0,15)
(13,8,25)
(34,50,45)
```

it would exit the factory with these coordinates:

```
(7,0,15)
(10,4,20)
(13,8,25)
(20,22,41.33333333333333)
(30,42,64.66666666666667)
(34,50,74)
```

In this second example, new vertices are added to ensure that no segment of any feature is longer than 10 units.

```
FACTORY_DEF SHAPE SamplingFactory \
INPUT FEATURE_TYPE * \
    @AddVertices(xy,10)
```

When the input feature from the previous example exits this factory, it would have these coordinates:

```
(7,0,15)
(13,8,25)
(17.4721359549996,16.9442719099992,35.434983894999)
(21.9442719099992,25.8885438199983,45.869967789998)
(26.4164078649987,34.8328157299975,56.3049516849971)
(30.8885438199983,43.7770876399966,66.7399355799961)
(34,50,74)
```

@Affine

@Affine(<A>, , <C>, <D>, <E>, <F>) for 2D and

@Affine(<A>, , <C>, <D>, <E>, <F>, <G>, <H>, <I>, <J>, <K>, <L>) for 3D

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<A>, , <C>, <D>, <E>, <F>	Real number, <A> and <E> must be non-zero.	Coefficients of the affine transformation. The transformation results in the x and y coordinates being modified by: $x' = Ax + By + C$ $y' = Dx + Ey + F$	No
<A>, , <C>, <D>, <E>, <F>, <G>, <H>, <I>, <J>, <K>, <L>	Real number, <A>, <F> and <K> must be non-zero.	Coefficients of the 3D affine transformation. The transformation results in the x,y and z coordinates being modified by: $x' = Ax + By + Cz + D$ $y' = Ex + Fy + Gz + H$ $z' = Ix + Jy + Kz + L$	No

Configuration

The @Affine function does not accept configuration lines.

Description

This function applies an affine transformation to the feature coordinates upon which it is invoked. An affine transformation not only preserves lines(2D) and planes(3D) it also preserves parallelism of lines and planes. That is, any lines or plane that were parallel before the transformation are parallel after the transformation. As well, if a number of points falling on a straight line or on a given plane are transformed, the resulting coordinates will fall on a straight line or a plane in the new coordinate system. Affine transformations include translations, rotations, scalings, and reflections.

A translation is a transformation that preserves the length, angle, and orientation of all geometric entities.

2D Translation : In this case, $A=E=1$, $B=D=0$, and C and F are the amounts of the translation

3D Translation : In this case, $A=F=K=1$, $B=C=E=G=I=J=0$, and D,H and L are the amounts of the translation

A rotation is a transformation that preserves the lengths and angles of all geometric entities. Rotations also preserve one point and the distance of all entities from that point.

Scaling transformations include those that preserve all angles and multiply all lengths by the same factor, thereby preserving the “shape” of all entities. Another form of scaling simply scales distances in the x direction by one amount, and distances in the y direction by another amount.

A reflection is a transformation that preserves lengths and magnitudes of angles but changes their sign. The effect is equivalent to viewing the original geometry in a mirror, or through a flipped-over sheet of transparent paper.

A special kind of affine transformation is called a shear. In a 2D shear that preserves horizontal lines, $A=E=1$, $D=0$, and B is not equal to zero. In a 2D shear that maintains vertical lines, $A=E=1$, $B=0$, and D is not equal to zero.

If the `FME_GEOMETRY_HANDLING` directive is set to “yes” in the mapping file, then arcs and ellipses will be preserved as such under a 2D transformation, and will be stroked to lines and polygons (respectively) in the case of a 3D transformation. If the directive is not present then only the center point of any arc or ellipse will be transformed.

Inverse Operation

If applied in the inverse direction, this function will apply the inverse of the affine transformation specified by its parameters.

In this case, A must be non-zero, as must $(A * E - D * B)$.

Note This operation is not supported for 3D features.

Example

In the example below, a shearing transformation is applied to all features as they flow through the FME:

```
FACTORY_DEF DWG SamplingFactory SAMPLE_RATE 1 \
INPUT FEATURE_TYPE * @Affine(1,0.5,0,0,1,0)
```

In this second example, a scaling transformation is applied to increase all x coordinates by a factor of two and increase all y coordinates by a factor of three.

```
FACTORY_DEF DWG SamplingFactory SAMPLE_RATE 1 \
INPUT FEATURE_TYPE * @Affine(2,0,0,0,3,0)
```

The following example shows how to perform 3D rotation around Z axis by 30 degrees:

```
FACTORY_DEF * TeeFactory \
    FACTORY_NAME Rotate_Z_30 \
    INPUT FEATURE_TYPE States_1 \
    OUTPUT FEATURE_TYPE Rotate_Z_30_AFFINED \
        @Affine(0.866,-0.5,0,0,0.5,0.866,0,0,0,0,1,0)
```


@Angle

```
@Angle(GEOM, <source type>, <dest type> [, <attr list>])
```

or

```
@Angle(ATTRIBUTES, <source type>, <dest type> [, <attr list>])
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<source type>	DECIMAL_DEGREES DDDMSS.SS0 DDDMSSSS DDDMSS RADIANS	The format of the input angles.	No
<dest type>	DECIMAL_DEGREES DDDMSSSS DDDMSS RADIANS	The format of the output angles after conversion.	No
<attr list>	attributes	The list of attributes containing angles that are to be converted. If GEOM is specified and this list is empty then only the coordinates will be converted <source type> from to <dest type>.	Yes

Configuration

The @Angle function does not accept configuration lines.

Description

The conversion type is one of the following:

DECIMAL_DEGREES	decimal degrees Example: 23.4567
DDMMSS.SSO	degrees, minutes, seconds Example: 274531.01N: 27 degrees, 45 minutes, 31.01 seconds North Note: This is not available as a destination type.

DDMMSSSS	degrees, minutes, hundredths of seconds Example: 78341034: 78 degrees, 34 minutes, 10.34 seconds
DDMMSS	degrees, minutes, seconds
RADIANS	radians Example: 1.456

If `GEOM` is specified, then the function converts the geometry coordinates from the `<source type>` to the `<dest type>` representation. If any attributes are specified in the `<attr list>` then the value of those attributes is also converted.

If `ATTRIBUTES` is specified, then only the attribute(s) in the `<attr list>` are converted from `<source type>` to `<dest type>`.

Inverse Operation

This function has no inverse.

Example

In the example below, the input feature's coordinates are converted from `DDMMSSSS` to decimal degrees along with the attributes `centroid_x` and `centroid_y`.

```

FACTORY_DEF * TeeFactory                                \
  INPUT FEATURE_TYPE *                                  \
  OUTPUT FEATURE_TYPE *                                 \
    @Angle(GEOM, DDMMSSSS, DECIMAL_DEGREES,            \
           centroid_x, centroid_y)

```

@ApplyRasterRotation

@ApplyRasterRotation()

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<interpolation type>	NEAREST-NEIGHBOR BILINEAR BICUBIC AVERAGE4 AVERAGE16	The type of interpolation to use. Default is NEARESTNEIGHBOR.	No

Configuration

This function does not accept configuration lines.

Description

Applies the raster rotation angle on the input raster properties to the rest of the raster properties and data values.

The expected input is a raster with a non-zero rotation angle and the expected output is be a rotated raster with a rotation angle of 0.0. It is expected that the input raster properties will be modified to conform the the output raster properties for a raster rotated by the given angle.

Applying a rotation angle is primarily done for compatibility with other processing and writers which cannot handle a rotation angle. It is suggested that the input raster also contain a nodata value since applying the rotation often has the effect of adding nodata areas around the corners of the rotated raster. These nodata areas will be filled with 0 or black values in the absence of an input raster nodata value.

Cell values are interpolated in order to rotate the raster; you have the choice of Nearest Neighbor, Bilinear, Bicubic, Average 4 or Average 16 interpolation methods. Nearest Neighbor is the fastest but produces the poorest image quality. Bilinear provides a reasonable balance of speed and quality. Bicubic is the slowest but produces the best image quality. Average 4 and Average 16 have a performance similar to Bilinear and are useful for numeric rasters such as DEMs.

Input features must contain raster geometries only.

This function is unaffected by input raster band and palette selection.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a typical use of the @ApplyRasterRotation function. In this example, the SamplingFactory is used to pass through every feature. Then, the @ApplyRasterRotation function is used to ensure that resultant features have a rotation of 0.0.

```
FACTORY_DEF * SamplingFactory          \  
    SAMPLE_RATE 1                      \  
    INPUT FEATURE_TYPE *               \  
        @ApplyRasterRotation(NEARESTNEIGHBOR) \  
    OUTPUT FEATURE_TYPE *
```

@Arc

```
@Arc(<semi-primary axis>,<semi-secondary axis>           \
    [, <numberOfEdges>, <rotation>, <startAngle>,       \
    <sweepAngle>, <centerX>, <centerY>])

or

@Arc(<direction>,<startX>,<startY>,<endX>,<endY>           \
    [, <numberOfEdges>])

or

@Arc([<numberOfEdges>])

or

@Arc([<numberOfEdges>], MAKEPOLY_FOR_360ARC)

or

@Arc(MAX_DEVIATION [,<maxDeviation>] [,MAKEPOLY_FOR_360ARC])
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<semi-primary axis>	Real Value	Length of the primary axis for the ellipse the arc is based upon.	No
<semi-secondary axis>	Real Value	Length of the secondary axis for the ellipse the arc is based upon.	No
<numberOfEdges>	Real Value	The number of edges in the final arc. The default is 0.	Yes
<rotation>	Real Value	The rotation of the ellipse that defines the arc. The rotation angle specifies the angle in degrees from the horizontal axis to the primary axis in a counterclockwise direction. Default is 0.	Yes
<startAngle>	Real Value	arctan((fme_primary_axis/fme_secondary_axis)*tan(realStartAngle)) Default is 0.	Yes

Name	Range	Description	Optional
<sweepAngle>	Real Value	$\arctan\left(\frac{fme_primary_axis}{fme_secondary_axis}\right) * \tan(realEndAngle)$ $\arctan\left(\frac{fme_primary_axis}{fme_secondary_axis}\right) * \tan(realStartAngle)$ Default is 360.	Yes
<centerX>	Real Value	The X coordinate of the center of the ellipse. If not specified, the first coordinate of the feature is used.	Yes
<centerY>	Real Value	The Y coordinate of the center of the ellipse. If not specified, the first coordinate of the feature is used.	Yes

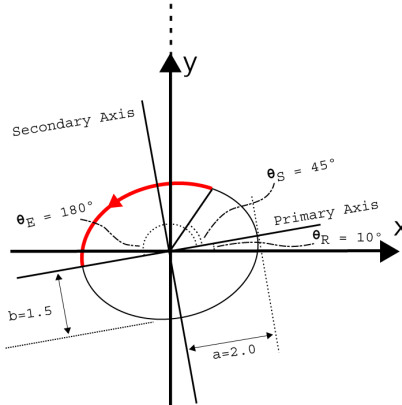
The following lists the arguments for the @Arc alternative that vectorizes a circular arc defined a the circle's center point, its arc start and end point, and its direction, either counterclockwise or clockwise, from start to end point.

Name	Range	Description	Optional
<direction>	String	Either CCW or CW (counterclockwise or clockwise), for the direction of the arc, from start to end point.	No
<startX>	Real Value	x coordinate value for the arc's start point	No
<startY>	Real Value	y coordinate value for the arc's start point	No
<endX>	Real Value	x coordinate value for the arc's end point	No
<endY>	Real Value	y coordinate value for the arc's end point	No
<numberOfEdges>	Real Value	The number of edges in the final arc. The default is 0.	Yes

The following lists the arguments for the @Arc alternative that vectorizes an arc defined by the attributes of the feature or the geometry itself.

Name	Range	Description	Optional
<numberOfEdges>	Real Value	The number of edges in the final arc. The default is 0.	Yes
MAKEPOLY_FOR_360ARC	N\A	This flag if present as the second argument will stroke arcs with 360 sweep angle into polygons. Otherwise, they are stroked into lines.	Yes

Name	Range	Description	Optional
MAX_DEVIATION	N\A	This flag if present as the first argument will stroke arcs using the <maxDeviation> value such that the distance between the arc and the vectorized line is no greater than the <maxDeviation>	No
<maxDeviation>	Real Value	The maximum deviation from the arc to the vectorized line.Default to 0	Yes



The Cartesian equation for the ellipse defines the curve by a simple parametric form with the x and y coordinates having different scalings:

$$\begin{aligned}x &= a \cos(t) \\y &= b \sin(t)\end{aligned}$$

where

a = primary radius
b = secondary radius

Note that t is a parameter which does not have a direct interpretation in terms of an angle. However, the relationship between the polar angle (theta) from the ellipse center and the parameter t follows from:

$$\begin{aligned}\theta &= \arctan((b/a) \cdot \tan(t)) \\ \text{or} \\ t &= \arctan((a/b) \cdot \tan(\theta))\end{aligned}$$

Therefore, to specify the red arc in the picture, in the counter clockwise direction, the following items would need to be set on the feature:

```
Coordinates should contain a single point, which is the center origin
fme_geometry_type = fme_point
fme_type = fme_arc
fme_primary_axis = a = 2.0
fme_secondary_axis = b = 1.5
fme_rotation = thetaR = 10 (degrees)
fme_start_angle = tS = arctan((a/b)*tan(thetaS)) = arctan((2.0/1.5)*tan(45))
                    = 53.130102354155978703144387440907
fme_sweep_angle = tE - tS = arctan((a/b)*tan(thetaE)) - tS = arctan((2.0/
                    1.5)*tan(180)) - tS = 180.0 - tS = 180.0 -
                    53.130102354155978703144387440907 =
                    126.86989764584402129685561255909
```

FIGURE 1-1 Sample Arc

Configuration

The @Arc function does not accept configuration lines. However, as described below, the FME keywords `FME_ARC_DEGREES_PER_EDGE` and `FME_ARC_EDGE_TOLERANCE` can be used to configure the behavior of @Arc.

Description

This function is often used when translating from formats supporting arcs and ellipses to those that do not by *vectorizing* such features.

This function replaces the geometry of the passed-in feature with a line, generated according to those parameters passed in. If the sweep angle of the arc is 360 degrees, then the resulting feature will be a closed polygon, shaped like an ellipse. Otherwise, the resulting geometry of the feature is a line.

If the number of edges is 0, then the FME will calculate the number of edges to create by dividing the `sweep_angle` by the setting of the mapping file directive `FME_ARC_DEGREES_PER_EDGE` which, by default, is 5. In addition, if the mapping file contains a setting for `FME_ARC_EDGE_TOLERANCE`, the generated line will be thinned to remove points that do not deviate more than this tolerance from the line connecting their neighbours. In this way, excessively large numbers of coordinate volumes are avoided when converting arcs into linestrings.

Alternatively, a circular arc defined by a circle's center point, its arc's start and end point, and its direction, either counterclockwise or clockwise from the start to end point can be vectorized by using the alternate `@Arc(<direction>, <startX>, <startY>, <endX>, <endY>)`, where the center point of the circle must be specified as the first coordinate of the feature.

If `MAX_DEVIATION` flag is present, then arcs will be vectorized such that the distance between the vectorized lines and the arcs is never greater than the `<maxDeviation>` value. The valid range for `<maxDeviation>` is (0, min{primary axis, secondary axis}). If `maxDeviation` is smaller than or equal to zero, the value of the mapping file directive

`FME_STROKE_MAX_DEVIATION`. If the value of this directive is also smaller than or equal to zero, then the arc will be vectorized using number of edges being zero. If the maximum deviation value is greater than or equal to the primary or secondary axis, then the arcs are vectorized using the minimum number of edges possible.

Inverse Operation

The function has no inverse.

Example

In the example below, arc features read from a Design file are changed to 50 segment lines while ellipse features are changed to 50 segment polygons during translation to Shape files.

```
IGDS * igds_type igds_arc \
  igds_primary_axis    %primary \
  igds_secondary_axis  %secondary \
  igds_rotation        %rotation \
  igds_start_angle     %start \
  igds_sweep_angle     %sweep \
SHAPE arcs \
  @Arc(%primary,%secondary,50,%rotation,%start,%sweep)
IGDS * igds_type igds_ellipse \
  igds_primary_axis    %primary \
  igds_secondary_axis  %secondary \
  igds_rotation        %rotation \
SHAPE ellipses \
  @Arc(%primary,%secondary,50,%rotation)
```

Tip

Since @Arc has no inverse, the above set of transfer specifications cannot be used to translate back from Shape files to a Design file.

@Area

@Area ([<multiplier>])

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<multiplier>	Real Number	By default, the area returned is in coordinate units squared. The multiplier, if specified, can be used to convert to other units. The default is 1.	Yes

Configuration

This function does not accept configuration lines.

Description

The @Area function calculates the area of a polygonal feature. The function correctly handles both polygonal features and polygonal features with holes. For point and linear features, 0 is returned. The optional multiplier can be used to convert the return value from ground units squared to units more useful to the caller.

Inverse Operation

The function has no inverse.

Example

In the example below, the Shape area attribute is set to the area of the SAIF Lake polygon object when features are translated from SAIF to Shape. The units are converted from square metres to hectares using a 0.0001 multiplication factor. When features are translated from Shape to SAIF, the @Area call has no effect.

```
SHAPE lake    area @Area(0.0001)
SAIF  Lake::MOF
```

@Bounds

```
@Bounds (<xMinAttr>, <xMaxAttr>, <yMinAttr>, <yMaxAttr> \
        [, <zMinAttr>, <zMaxAttr>] \
        [, GEOMETRIC] )
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<xMinAttr>	String	The name of the attribute assigned the minimum value of the feature along the X axis.	No
<xMaxAttr>	String	The name of the attribute assigned the maximum value of the feature along the X axis.	No
<yMinAttr>	String	The name of the attribute assigned the minimum value of the feature along the Y axis.	No
<yMaxAttr>	String	The name of the attribute assigned the maximum value of the feature along the Y axis.	No
<zMinAttr>	String	The name of the attribute assigned the minimum value of the feature along the Z axis.	Yes
<zMaxAttr>	String	The name of the attribute assigned the maximum value of the feature along the Z axis.	Yes
GEOMETRIC	String	If specified, the geometric bounding box will be returned.	Yes

Configuration

The @Bounds function does not accept configuration lines.

Description

This function extracts the bounds of a feature. It determines the extreme values of the feature in each of the x and y (and optionally z) axes, and assigns these values to the attribute names passed in as arguments.

Normally the `@Bounds` function considers the bounding box of the feature to be the bounding box of all the coordinates in the feature. If the last argument to the function is the string `GEOMETRIC`, the function will return the geometric bounding box of the feature, which in the case of some point features may be different. For example, text features only have a single coordinate (to specify their location), so normally their bounding box only contains that single point. However, if the `GEOMETRIC` flag is specified, the returned bounding box will contain the entire feature, as if it were rendered with a fixed-width font at its given height. Arcs and ellipses are treated similarly.

When using the `GEOMETRIC` flag, please note that the generic form of any geometry attributes are required to exist on the feature in order to properly determine the bounds. For example, a text feature should have an `fme_type` attribute with a value of `fme_text`, and other geometry attributes that describe a text geometry such as `fme_text_string`, should also be present.

Example

This function has no inverse.

In the example below, each feature is passed through a `SamplingFactory`. As each feature enters the factory, it has six new attributes added to it to hold the extents of the feature.

```
FACTORY_DEF SDE SamplingFactory
    SAMPLE_RATE 1
    INPUT FEATURE_TYPE *
@Bounds(xmin,xmax,ymin,ymax,zmin,zmax)
```

If a feature had the coordinates (1,10,100), (2,-20,150), then after leaving the factory, it would have these new attributes with these values:

Attribute	Value
xmin	1
xmax	2
ymin	-20
ymax	10
zmin	100
zmax	150

@Buffer

```
@Buffer(<bufferWidth> [, [<strokeAngle> [, <bufferStyle>]]])
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<bufferWidth>	Real >= 0	The distance the buffer boundary will be from the original geometry, measured in ground units.	No
<strokeAngle>	Real >= 0	The angle between successive approximations of a circular curve, measured in degrees. If 0 is specified, then no curves will be used to smooth the buffer. The default is 5.	Yes
<bufferStyle>	CAP_ROUND CAP_ROUND:<deg> CAP_BUTT CAP_PROJECTING SIDES_ONLY LEFT_SIDE_ONLY RIGHT_SIDE_ONLY	This sets the buffer or ending style. The default is CAP_ROUND.	Yes

Configuration

The @Buffer function does not accept configuration lines.

Description

This function replaces the geometry of a feature with a geometry representing a buffer of specified width around the original feature. If the width of the buffer is nonzero, the result will be a polygon or donut polygon. Each point in the boundary of the returned area will be <bufferWidth> distance from the original feature. (If one of the side <bufferStyle> options are chosen, the output will be a line or aggregate.)

The <strokeAngle> controls how smooth any curves in the resulting buffer polygons are. Measured in degrees, vertices approximating the curve will be generated at this interval. The value 0 is special and causes no curve approximations to be done.

If the feature was a point, it is replaced by a circle with a radius of `<bufferWidth>`.

Each line is rereplaced by a polygon representing a buffer of width `<bufferWidth>` on either side of the line.

An area feature is treated as a set of the lines which make up its boundaries. Each boundary will be buffered as if it were a line.

If the feature was an aggregate feature, a single buffer is computed from all members of the aggregate.

The `<bufferStyle>` controls other aspects of how the returned buffer will appear.

bufferStyle	Description
CAP_ROUND	This rounds the ends. It uses the same rounding angle as set with <code><strokeAngle></code> . This is the default.
CAP_ROUND: <deg>	This rounds the ends, but uses a different stroke angle. Example <code>CAP_ROUND: 25</code>
CAP_BUTT	This makes the ends of buffered lines flat, with the end of the original line touching the end of the buffer.
CAP_PROJECTING	This makes the ends of buffered lines flat, with the buffer extending out <code><bufferWidth></code> units beyond the end of the original line.
SIDES_ONLY	The result is not a polygon, but an aggregate which is both the left and right sides of the buffer as lines.
LEFT_SIDE_ONLY	The result is not a polygon, but is just the left side of the buffer as a line. In the case where the line being buffered forms a closed area, this reverts to <code>CAP_ROUND</code> behavior.
RIGHT_SIDE_ONLY	The result is not a polygon, but is just the right side of the buffer as a line. In the case where the line being buffered forms a closed area, this reverts to <code>CAP_ROUND</code> behavior.

Note This function is two-dimensional. Three-dimensional features will result in two-dimensional output buffered features, and z coordinates will be lost.

Inverse Operation

This function has no inverse.

Example

In the example below, linear `Pipeline` features are replaced by polygons which buffer them by 10 ground units.

```
FACTORY_DEF SHAPE SamplingFactory \
SAMPLE_RATE 1 \
INPUT FEATURE_TYPE Pipeline @Buffer(10,22.5)
```


@Buffer2

Note: The @Buffer2 function uses functionality from the GEOS library (<http://geos.refractions.net/>), which is distributed under the terms of the Free Software Foundation's LGPL license (<http://www.gnu.org/licenses/lgpl.html>). In order to comply with the terms of the LGPL, Safe Software Inc. has set up a website (<http://www.safe.com/foss>) to provide further information and to distribute source code as required.

```
@Buffer2(<bufferWidth> [, [<strokeAngle> [, <bufferStyle>]]])
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<bufferWidth>	Real	The distance the buffer boundary will be from the original geometry, measured in ground units.	No
<strokeAngle>	Real >= 0	The angle between successive approximations of a circular curve, measured in degrees. If 0 is specified, then no curves will be used to smooth the buffer. The default is 5.	Yes
<bufferStyle>	CAP_ROUND CAP_ROUND:<deg> CAP_BUTT CAP_PROJECTING SIDES_ONLY LEFT_SIDE_ONLY RIGHT_SIDE_ONLY	This sets the buffer or ending style. The default is CAP_ROUND.	Yes

Configuration

The @Buffer2 function does not accept configuration lines.

Description

This function replaces the geometry of a feature with one that represents the original, padded by a specified width parameter, <bufferWidth>. Each point in the output geometry will be at a distance <bufferWidth> from the original geometry.

The parameter <strokeAngle>, measured in degrees, controls the smoothness of curves in the resulting buffer. Vertices approximating the curve will be

generated at this interval. If the value 0 is specified, no curve approximations are made and corners are beveled.

There are two main buffering styles, regular buffering and side buffering, specified by the parameter `<bufferStyle>`.

Regular buffering styles include `CAP_ROUND`, `CAP_BUTT`, and `CAP_PROJECTING`. These styles apply to all geometries, and always output area features (i.e. polygon or donut features). The cap style describes the type of ending for input line features. If the input feature is an area, the particular cap style is ignored.

If a zero `<bufferWidth>` is specified, then the output features will not grow or shrink in size; all geometries of this feature are dissolved together. For example, a feature containing an aggregate geometry, using `@Buffer2` with a width of 0.0, will generate a dissolved version of the feature.

If a negative `<bufferWidth>` is specified, then the output features will be shrunk. Please note that when specifying a negative `<bufferWidth>` for input line features, the output will result in null geometries for those line features.

The following describes the different types of end cap styles available:

bufferStyle	Description
<code>CAP_ROUND</code>	This style rounds line endings by a stroked arc. The smoothness of the round cap can be controlled by the parameter <code><strokeAngle></code> . <i>This is the default setting.</i>
<code>CAP_ROUND: <deg></code>	This style is like <code>CAP_ROUND</code> , except a custom angle parameter <code><deg></code> can be specified.
<code>CAP_BUTT</code>	This style makes line endings flush to the end of the line, that is, the end of the line touches the end of the buffer.
<code>CAP_PROJECTING</code>	This style creates square line endings. The buffer extends beyond the line ending by a width of <code><bufferWidth></code> .

Note This function is two-dimensional. Three-dimensional features will result in two-dimensional output buffered features, and z coordinates will be lost.

Side buffering styles include `SIDES_ONLY`, `LEFT_SIDE_ONLY`, and `RIGHT_SIDE_ONLY`. These styles apply to line features only - features with geometries that do not contain lines will generate no output geometries. Unlike regular buffering, side buffering styles can only accept positive values of `<bufferWidth>`.

Side buffering styles will only generate valid output geometries if each input line feature does not self-intersect.

The following describes the different types of side buffering available:

bufferStyle	Description
LEFT_SIDE_ONLY	This style generates a line representing the left side of a regular buffer.
RIGHT_SIDE_ONLY	This style generates a line representing the right side of a regular buffer.
SIDES_ONLY	This style generates lines representing both the left and right sides of a regular buffer.

If the `FME_GEOMETRY_HANDLING` directive is set to yes in the mapping file, arcs and ellipses will be stroked prior to being buffered; they are otherwise buffered as points located at their respective center points.

If the `FME_BUFFER2_DIVIDE_AND_CONQUER` directive is set to yes or is undefined in the mapping file, buffering of aggregate geometries will be performed using a divide and conquer algorithm. If this directive is set to no, buffering is performed against the entire collection at once. The divide and conquer algorithm splits the collection in to smaller pieces, buffers the individual pieces, then unions the pieces back together. The divide and conquer algorithm is the default, since it has higher performance and lower resource requirements than buffering at once. Buffering at once can be advantageous in specific situations, as it is more robust when dealing with degenerate geometries.

Inverse Operation

This function has no inverse.

Example

In the example below, linear `Pipeline` features are replaced by polygons which buffer them by 10 ground units.

```

FACTORY_DEF SHAPE SamplingFactory \
SAMPLE_RATE 1 \
INPUT FEATURE_TYPE Pipeline @Buffer2(10,22.5)

```


|@CheckpointRaster

| @CheckpointRaster()

Function Type: Feature

| **Arguments: None**

Configuration

This function does not accept configuration lines.

Description

The @CheckpointRaster function is used to force the previous processing to occur immediately and to save the state of the processing to disk. This is an advanced function for managing raster performance.

Input features must contain raster geometries only.

This function is unaffected by input raster band and palette selection.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a typical use of the @CheckpointRaster function. In this example, the SamplingFactory is used to pass through every feature. Then, the @CheckpointRaster function is used to save the state of each raster for future processing.

```

FACTORY_DEF * SamplingFactory                                \
    SAMPLE_RATE 1                                           \
    INPUT FEATURE_TYPE *                                    \
        @CheckpointRaster()                                \
    OUTPUT FEATURE_TYPE *
```

@Circularity

@Circularity()

Function Type: Attribute

Arguments: None

Configuration

The @Circularity function does not accept configuration lines.

Description

This function is responsible for computing a circularity measure for the passed-in feature. The feature should be area-based, though the function returns a value that is not particularly meaningful for line and point features. For a point feature, a value of 1 is returned. For a linear feature, a value of 0 is returned.

The function computes how elongated a polygonal feature is. The measure returned is:

$$4 * \text{PI} * \text{area} / (\text{perimeter} * \text{perimeter})$$

Tip

This function is often combined with the `TestFactory` to select features for area generalization operations.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

In the example below, the circularity is stored as an attribute named `circularity` when going from Shape to MIF. The function has no effect when invoked in the opposite direction.

```
SHAPE lakes
MIF lakes circularity @Circularity()
```


@Close

@Close()

Function Type: Feature

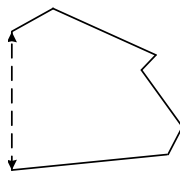
Arguments: None

Configuration

The @Close function does not accept configuration lines.

Description

This function takes a feature with linear geometry and closes it by connecting the first point with the last point. It is used to artificially close polygons with an edge missing. For example, some data sets do not include the map neat line, so any areas abutting the map edge are not closed. @Close is useful in these situations.



WARNING! Use this function with caution – in most situations simply connecting the first point with the last point will not produce a valid polygon.

If @Close is invoked on a feature that is already a polygon, it will do nothing. If the feature was tagged as being a line and its first and last points are the same, then this function will simply adjust the feature's `fme_geometry` to be `fme_polygon` and no change is made to the geometry.

Inverse Operation

This function has no inverse.

Example

In the example below, the `PolygonFactory` is used to form polygons from the `LandCover` boundaries present in a SAIF input file. Any boundaries that did not close are output from the `PolygonFactory` according to the `OUTPUT LINE` clause in the factory definition. The `@Close` function is invoked on each such unclosed linear feature as it emerges from the factory, turning it into a polygon.

```

FACTORY_DEF SAIF PolygonFactory                                \
  INPUT      FEATURE_TYPE LandCover::TRIM                     \
              position.geometry.Class OrientedArc             \
GROUP_BY    landCoverType                                     \
OUTPUT LINE  FEATURE_TYPE LandCover::TRIM                     \
              position.geometry.Class Polygon                 \
              @Close()                                         \
OUTPUT POLYGON FEATURE_TYPE LandCover::TRIM                   \
              position.geometry.Class Polygon

```

@Concatenate

```
@Concatenate(<string>[,<string>]*)  
or  
@Concatenate(<list>,<separator>)
```

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<string>	Any String	All strings passed in are joined together to form a single, composite string, which is returned. A carriage return can be specified by using \n as the string parameter.	No
<list>	Any Feature Attribute List	The list whose elements are to be concatenated. The name should contain a {} pair to indicate that it is a list.	No
<separator>	Any Single or Multiple Character	The character string used as a separator between the concatenated elements of the list. If a comma is desired, it must be escaped (otherwise it will be considered a parameter separator). Other characters in the speparator may also be escaped.	No

Configuration

The @Concatenate function does not accept configuration lines.

Description

When this function is passed a set of strings as parameters, it concatenates the strings, using no separation characters, and returns the result as a single string.

If any parameter starts with a backslash (“\”), it is interpreted as a quoted special character sequence, as specified in the following table. If the sequence is not listed in the table, then the backslash character is simply ignored. This substitution is only done when the complete parameter is one of these character sequences; if additional characters are part of the parameter, then no substitution is done.

Sequence	Description
\a	Audible alert (bell) (0x07)
\b	Backspace (0x08)

Sequence	Description
\f	Form feed (0x0c)
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\t	Tab (0x09)
\v	Vertical tab (0x0b)
\\	Backslash
\/	Forward slash
\ooo	The digits ooo describe a character in octal notation. (There may be one, two, or three octal digits.)
\xhh	The digits hh describe a character in hexadecimal notation. (There may be one or two hexadecimal digits.)

When this function is passed a list attribute and a separator, it concatenates the elements of the list, places the separator character between each pair of elements, and returns the result as a single string. The separator character may also be expressed as a special character sequence beginning with a backslash, as described in the above table.

Tip

@Concatenate is similar to the inverse of the @Split function. @Split is useful in transfer specifications where an inverse is required. @Concatenate is convenient in conjunction with factories when no inverse is needed.

When this function encounters the special token `fme_attribute_name`, the next token it receives is treated as an attribute name. In this case, the value of the attribute whose name is passed in will be concatenated, rather than the name of the attribute.

Inverse Operation

This function has no inverse.

Example 1

In the example below, the values of the `street` and `city` attributes are joined together separated by a hyphen and the result is assigned to the `address` attribute.

```

FACTORY_DEF SHAPE TeeFactory \
    INPUT FEATURE_TYPE Building \

```

```

        address @Concatenate(&street, " - ", &city)
OUTPUT FEATURE_TYPE *
\

```

Example 2

In the example below, if the `LNAM_REFS{}` list attribute is not empty, its elements are joined together separated by a comma and the result is assigned to a new `LNAM_REFS` attribute.

```

FACTORY_DEF * TestFactory
INPUT FEATURE_TYPE *
    TEST &LNAM_REFS{0} == ""
    OUTPUT PASSED FEATURE_TYPE *
    OUTPUT FAILED FEATURE_TYPE *
        @SupplyAttributes(LNAM_REFS, @Concatenate(LNAM_REFS{ }, "\, "))
\
\
\
\
\

```

Example 3

In the example below, the values of the `city` and `state` attributes are together separated by a carriage return and the result is assigned to the `label` attribute.

```

FACTORY_DEF SHAPE TeeFactory
INPUT FEATURE_TYPE Building
    label @Concatenate(&city, "\n", &state)
OUTPUT FEATURE_TYPE *
\
\
\

```

Example 4

In the example below, the value of the `street_name` attribute will be concatenated to `street`.

```

@Concatenate(&street, fme_attribute_name, street_name)

```

@ConvertBase

```
@ConvertBase(<value>,<originalBase>,<destBase>
              [,<outputWidth>])
```

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<value>	String ^a	The string representation of the number to be converted from the <originalBase> to the <destBase>. The number is expressed in <originalBase> notation, meaning it only uses the first <originalBase> characters from the set as its digits.	No
<original Base>	2..36	The base of the <value> parameter.	No
<destBase>	2..36	The base to which the value is to be converted.	No
<output Width>	<decimal number>	The desired number of digits in the output number. The output number is padded on the left with zeros to fill it out to the desired width. If the output number has more digits before padding than the width, it will not be trimmed but will be returned untouched.	Yes

a. Use only the first <originalBase> characters from the set:

```
0123456789 ABCDEFGHIJKLMNOPQRSTUVWXYZ.
```

Configuration

The @ConvertBase function does not accept configuration lines.

Description

This function converts a value expressed in one base to another, and returns the result. The @ConvertBase function only supports unsigned whole numbers. It does **not** handle fractions or decimal points – only **unsigned** integer numbers are supported. Any base, from base 2 (binary) to base 36, is supported. Digits are chosen from the set:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Any lowercase digits are automatically converted to uppercase by this function. An optional output width may also be specified. If the converted value has fewer

digits than the width, then it will be padded on the left with zeroes to fill out the width. If the converted value has more digits than the width, nothing will be done. This function has support for very large numbers.

Inverse

The inverse of this function does the reverse base conversion from the specified <destBase> to the <originalBase>. No padding is done in this case.

Example

The following call to @ConvertBase converts the number 255 from decimal (base 10) to hexadecimal (base 16):

```
@ConvertBase(255,10,16)
```

and returns the value FF.

This call does the same thing, but pads the result to four digits:

```
@ConvertBase(255,10,16,4)
```

and returns the value 00FF.

More commonly however, @ConvertBase operates on the value of an attribute rather than on a hard-coded constant. In this example, the 32 digit hex string returned by the @GOID (Geographic Object Identifier) function is split into four eight-digit hex numbers, which are then each converted to decimal and assigned to attributes.

```
# First compute the GOID, assigning it to the "hexGoid"
# attribute.
# Then split the hexGoid into four eight-digit strings
FACTORY_DEF SAIF TeeFactory                                     \
    INPUT FEATURE_TYPE *                                       \
    OUTPUT FEATURE_TYPE * hexGoid @Goid()                     \
        @Split(&hexGoid,"4s4s4s4s",hexPt1,hexPt2,hexPt3,hexPt4)

# Now convert the four eight-digit hex strings into their
# decimal equivalent
FACTORY_DEF SAIF TeeFactory                                     \
    INPUT FEATURE_TYPE *                                       \
    OUTPUT FEATURE_TYPE *int1 @ConvertBase(&hexPt1,16,10) \
        int2 @ConvertBase(&hexPt2,16,10) \
        int3 @ConvertBase(&hexPt3,16,10) \
        int4 @ConvertBase(&hexPt4,16,10)
```

After leaving these two factories, each feature has four unsigned integer attributes available to be stored in 32-bit unsigned integers in the output format.

@ConvertToArc

```
@ConvertToArc (<primaryRadiusAttrName>,           \
               <secondaryRadiusAttrName>,         \
               <startAngleAttrName>,               \
               <sweepAngleAttrName>)
```

or

```
@ConvertToArc (<primaryRadiusAttrName>,           \
               <secondaryRadiusAttrName>,         \
               <startAngleAttrName>,               \
               <sweepAngleAttrName>,               \
               <direction>,                         \
               <startX>, <startY>,
```

```
<endX>, <endY>)
```

or

```
@ConvertToArc (ARC3POINTS)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<primary RadiusAttr Name>	String	The name of the attribute set to the radius of the arc along the primary axis.	No
<secondary RadiusAttr Name>	String	The name of the attribute set to the radius of the arc along the secondary axis. ^a	No
<startAngle AttrName>	String	The name of the attribute set to the angle of the arc, as measured counterclockwise from horizontal.	No
<sweepAngle AttrName>	String	The name of the attribute set to the sweep, or duration, of the arc. This is always a positive value and is counterclockwise from the starting angle.	No

The following arguments are ONLY required for the second alternative of the function.

<direction>	String	CCW or CW, for counterclockwise or clockwise direction.	No
-------------	--------	---	----

Name	Range	Description	Optional
<startX>	Real value	x coordinate value for the start point of the circular curve.	No
<startY>	Real value	y coordinate value for the start point of the circular curve	No
<endX>	Real value	x coordinate value for the end point of the circular curve	No
<endY>	Real value	y coordinate value for the end point of the circular curve	No
The following argument is applicable ONLY to the @ConvertToArc(ARC3POINTS) alternative. It builds an arc definition where the 3 coincident points on the circular arc specified in the coordinates of a linear feature.			
ARC3POINTS	String	A constant value.	No

- a. In this FME release, only circular arcs are generated; therefore, the primary and secondary radii are **always** the same.

Configuration

The @ConvertToArc function does not accept configuration lines.

Description

This function calculates the center point, radius, start angle, and sweep angle of a circular arc that starts at the first point of the line passed in, ends at the last, and passes through the *middle* point where middle is the (n/2)th point in the line, not geometrically the middle point. It sets attributes on the feature to hold the radius, start angle, and sweep angle, and sets the geometry of the feature to be a single point that is the center point of the circle.

If this function is called on a feature with less than 3 points, it will abort the translation. If this function is called on a feature with exactly 3 points, the arc will be guaranteed to pass through all of them. If it is called on a feature with more than 3 points, the arc will approximate the curve and will be guaranteed to pass through the first, last, and *middle* point only.

Alternatively, the radius, start angle, and sweep angle can be calculated from a circular arc defined by the circle’s center point, the arc’s start point and end point, and finally the direction of the arc, either counterclockwise or clockwise from start to end. The center point of the circle must be specified as the first coordinate of the feature. There is no inverse for this @ConvertToArc alternative.

`@ConvertToArc (ARC3POINTS)` must be applied to a linear feature where the coordinates of the line correspond to the 3 control points coincident to the circular arc. `ARC3POINTS` is a constant value. The number of points in the line must be at least 3, if the number of points in the line are greater than 3, then the number of points must be odd, specifically they must be, $(2*N + 1)$, where N is the number of consecutive arcs formed. Each consecutive arc share a common control point, If more than one arc is formed then the feature will be converted into a path with consecutive arc segments. There is no inverse for this `@ConvertToArc (ARC3POINTS)`.

Inverse

When called on the source line of a transfer specification in the inverse direction, this function converts a point feature into a linestring that approximates an arc defined by the values of the parameter names passed to it. For detailed documentation on the `@Arc` function, see the `@Arc` section in this chapter.

Example 1

Given a feature containing these points

```
2,1
1,2
0,√5
```

a call to

```
@ConvertToArc (p_radius,s_radius,startAng,sweepAng)
```

turns the feature into a point feature with this geometry

```
0,0
```

and these attributes

```
p_radius √5
s_radius √5
startAng 30
sweepAng 60
```

Example 2

When arc features are reprojected, FME automatically converts them into linestrings. However, sometimes there is a requirement to maintain the features as arcs even after they have been reprojected.

The following `SamplingFactory` provides the solution for such a case. It accepts only arc features read in from an input Design (IGDS) file. As each feature enters the factory, it is first reprojected. This turns the feature into a linestring. Then the feature is turned back into an arc using `@ConvertToArc`.

```

FACTORY_DEF MIF SamplingFactory \
SAMPLE_RATE 1 \
INPUT FEATURE_TYPE * igds_type igds_arc \
@Reproject(UTM10-27,UTM10-83) \
@ConvertToArc (igds_primary_axis, \
               igds_secondary_axis, \
               igds_start_angle, \
               igds_sweep_angle)

```

Example 3

Given a feature with coordinate point 0,0 (the center point of the circle), the radius (primary and secondary are the same), the start angle and the sweep angle can be calculated for an arc going in the counterclockwise direction from the start point (0.866025,0.5) to the end point (0.5,0.866025) with

```

@ConvertToArc(p_radius,s_radius,startAng,sweepAng,\
CCW,0.866025,0.5,0.5,0.866025)

```

Example 4

Given a feature a linear feature with coordinates, (100,500), (200,600), (300,500), (500,700), and (700,500) the @ConvertToArc (ARC3POINTS) function converts the line into a path containing two circular arcs. The first arc in the path is defined by the start-point: (100,500), mid-point: (200,600) and end-point: (300,500), and the second arc is defined by the start-point: (500,700), mid-point: (500,700) and end-point: (700,500).

@ConvertToLine

Note: The @ConvertToLine function (in MEDIAL_AXIS or SKELETON mode) uses an algorithm designed by Petr Felkel and Stepan Obdrzalek, which is provided as-is, with no warranties. The paper outlining this algorithm is referenced in Straight Skeleton Implementation, 1998, "SCCG 98: Proceedings of the 14th Spring Conference on Computer Graphics," pages 210-218, ISBN 80-223-0837-4 (Publisher: Comenius University, Bratislava).

```
@ConvertToLine(MEDIAL_AXIS)
```

```
@ConvertToLine(SKELETON)
```

```
@ConvertToLine(CLASSIC, <tolerance>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<tolerance>	Any real number greater than 0.	The tolerance that defines the point density on the resulting line. The resulting lines do not have points closer than the specified tolerance.	No

Configuration

The @ConvertToLine function does not accept configuration lines.

Description

When called with an argument of MEDIAL_AXIS or SKELETON, the geometry of the feature is replaced by its medial axis or straight skeleton ("angular bisector network"), respectively. The medial axis is the subset of the straight skeleton that does not include any edges sharing a vertex with the original area.

Note In either of these modes, the algorithm may take a long time to run on large input features. The efficiency is given as $O(nm + n \log m)$, where n is the total number of vertices, and m is the number of "reflex vertices" – vertices causing the polygon to be non-convex.

In CLASSIC mode, the function replaces the geometry of the passed-in feature with a line. The line is generated by threading through the center of the polygonal shape. This mode requires a single parameter: the tolerance. Lines generated have points no closer than this distance apart. This function, when

combined with the `TestFactory`, enables FME to perform area generalization operations.

Unexpected output may result if input polygons self-intersect or have duplicate vertices. Also, because Z coordinates are not considered in either algorithm, all features processed by this function are forced to 2D.

If non-area features are passed to this function, no changes will be made to them and they are logged with a warning.

Inverse Operation

This function has no inverse.

Example

In the example below, polygonal `RoadAllowance` features with areas less than 1000 square ground units and a circularity measure less than 0.5 are converted from polygons to lines. All other `RoadAllowance` features are untouched. Notice the use of the `@Evaluate` and `@Circularity` functions to perform the test.

```
FACTORY_DEF SHAPE TestFactory                                \
  INPUT FEATURE_TYPE RoadAllowance                          \
  TEST @Evaluate("(@Area()<1000)&&(@Circularity() < 0.5)") = 1 \
  OUTPUT PASSED FEATURE_TYPE RoadLine                        \
    @ConvertToLine(10)                                       \
  OUTPUT FAILED FEATURE_TYPE RoadAllowance
```

@ConvertToPoint

```
@ConvertToPoint( [ BOUNDING_BOX | CENTER_OF_GRAVITY ] )
```

Function Type: Feature

Arguments: None.

Configuration

The @ConvertToPoint function does not accept configuration lines.

Description

This function replaces the geometry of the passed-in feature with a point. The point is defined to be either the center of the bounding box or the center of gravity of the feature. This function, when combined with the `TestFactory`, enables the FME to perform area generalization operations.

The optional parameter specifies how @ConvertToPoint() is to compute the point. A value of `CENTER_OF_GRAVITY` tells it to use the center of gravity of the feature's geometry as the point, and a value of `BOUNDING_BOX` tells it to use the bounding box. If the parameter is not specified, a value of `BOUNDING_BOX` is assumed.

Inverse Operation

This function has no inverse.

Example

In the example below, polygonal `Building` features with areas less than 1000 square ground units are converted from polygons to points, and features larger than this are untouched.

```
FACTORY_DEF SHAPE TestFactory                                \
  INPUT FEATURE_TYPE Building                                \
  TEST @Area() < 1000                                        \
  OUTPUT PASSED FEATURE_TYPE BuildingPoint                   \
    @ConvertToPoint()                                         \
  OUTPUT FAILED FEATURE_TYPE Building
```

@ConvexHull

@ConvexHull()

Function Type: Feature

Arguments: None.

Configuration

The @ConvexHull function does not accept configuration lines.

Description

This function replaces the geometry of the passed-in feature with a polygon representing its convex hull. The convex hull is defined as the minimum enclosing convex polygon. A convex polygon is a polygon where no interior angle is greater than 180 degrees. In lay terms, the effect is similar to tightening a rubber band around the feature.

If the feature was an aggregate feature, a single convex hull is computed from all vertices of all geometries in the aggregate. If the feature was a linear feature, it will be turned into a polygonal feature. No change is made to a point feature.

If the `FME_GEOMETRY_HANDLING` directive is set to “yes” in the mapping file, arcs and ellipses will be stroked prior to the calculation of the convex hull; they are otherwise regarded as points located at their respective center points.

Inverse Operation

This function has no inverse.

Example

In the example below, linear `Pipeline` features are replaced by their convex hulls.

```
FACTORY_DEF SHAPE SamplingFactory \
SAMPLE_RATE 1 \
INPUT FEATURE_TYPE Pipeline @ConvexHull()
```

@Coordinate

```
@Coordinate( (x|y|z), (<index>|END), [FLATTEN_AGGREGATE])
```

```
@Coordinate( (KEEP|REMOVE), (<index>|END), [<length>])
```

```
@Coordinate(RESET_FROM_ATTRIBUTE, xattr, yattr [,zattr])
```

Function Type: Attribute and Feature

Arguments:

Name	Range	Description	Optional
<axis>	(x y z)	The first parameter controls the axis from which the coordinate value are taken.	No
<index>	0..<@NumCoords()-1> -<@NumCoords()-1> END	The second parameter indicates which coordinate is of interest. The first coordinate is given the index 0, and the last coordinate is at index position @NumCoords()-1. As a convenient shortcut, the last coordinate of the feature may be accessed by using END as the index. As well, a negative index can be used to count the coordinates from the end of the feature. -1 refers to the last coordinate, -2 the second last, and so on.	No
<length>	0..<@NumCoords()-1>	The third parameter indicates number of coordinates in the range of interest, starting at the position indicated in the second parameter.	Yes
FLATTEN_AGGREGATE N\A		This flag, if present, will ensure that the coordinate returned for multi-part feature(aggregate feature) will not include the internal FME meta information for aggregates.	Yes

Name	Range	Description	Optional
RESET_FROM_ATTRIBUTE	N\A	This flag if present as the first argument will allow to set the coordinate values from the attributes on the feature	No
xattr	Existing attribute name	Name of the attribute that will be used to get the x value for the coordinate	No
yattr	Existing attribute name	Name of the attribute that will be used to get the y value for the coordinate	No
zattr	Existing attribute name	Name of the attribute that will be used to get the z value for the coordinate	Yes

Configuration

The @Coordinate function does not accept configuration lines.

Description

The first form of calling this function returns the value of a single coordinate of a feature.

If the feature is 2D and a request is made for a value on the third axis, zero will be returned.

If the given index is out of range, then an exception will be raised and the translation will abort.

If the first parameter is `KEEP`, then this function will only keep those coordinates within the specified range. If the first parameter is `REMOVE`, then this function will remove all the coordinates in the specified range. If `KEEP` or `REMOVE` is attempted on a feature that is an aggregate or solid or surface, then an exception is raised and the translation is aborted.

The second parameter indicates the index of the first coordinate in the range. The index may be any integer from zero to one less than the number of coordinates in the feature, or a negative number that specifies the index measured from the end of the feature. -1 refers to the last coordinate, -2 the second last, and so on. If `END` is passed in, then the last coordinate is the initial coordinate of the range.

If the index given is out of range, then an exception will be raised and the translation will abort.

The third parameter is the number of coordinates in the range, starting at the position indicated in the second parameter. If the third parameter is not given, a value of "1" is assumed. If the third parameter is not an integer or is negative, the translation will abort.

If the number of coordinates indicated extends beyond the number of coordinates that exist, the range will be considered as extending from the second parameter to the last coordinate (without complaint).

For both `KEEP` and `REMOVE`, the value returned is the actual number of coordinates removed, if any.

If the first parameter is `RESET_FROM_ATTRIBUTE` then the subsequent parameters will be attribute names from which the x, y and optionally z values will be obtained for setting the coordinates on the feature. The required second and third parameter `<xattr>` and `<yattr>` should be the name of attributes from which the values of x and y coordinates will be retrieved.

All features that end up as a single point will have their `fme_geometry` set to `fme_point`.

All features that end up with no geometry will have their `fme_geometry` set to `fme_undefined`.

Features that end up with more than one point will have their `fme_geometry` unchanged. The user should be careful that this remains valid.

Inverse

The function performs the same operation in the inverse direction.

Example

In the example below, the x, y, and z coordinates for the first and last points in the feature are extracted and stored in the attributes `xFirst`, `yFirst`, `zFirst`, `xLast`, `yLast`, and `zLast`. These attributes may then be used in further calculations.

The `TestFactory` is used to separate single points from multi-point lines.

```

FACTORY_DEF SAIF TestFactory                                \
  INPUT FEATURE_TYPE *                                      \
  TEST @NumCoords() > 1                                     \
  OUTPUT PASSED FEATURE_TYPE *                              \
    xFirst @Coordinate(x,0)                                  \
    yFirst @Coordinate(y,0)                                  \
    zFirst @Coordinate(z,0)                                  \
    xLast  @Coordinate(x,END)                                 \
    yLast  @Coordinate(y,END)                                 \
    zLast  @Coordinate(z,END)                                 \

```


@CoordSys

```
@CoordSys ([coordsys-name])
```

```
@CoordSys (__CONVERT_COORDSYS_NAME__,  
<inputCoordsysAttr>, <CONVERSION_DIRECTION>,  
<EXTERNAL_SYSTEM_NAME>, <outputCoordsysAttr>)
```

Function Type: Feature OR Attribute

Arguments:

Name	Range	Description	Optional
<coordsys-name>	coordinate system name	The name of the coordinate system for the feature to set.	Yes, when the function is being used as an attribute function.
__CONVERT_COORDSYS_NAME__	N/A	This flag if present as the first argument indicates that the function will be used for converting FME coordinate system name into its equivalent representation in the external system and back	No
<inputCoordsysAttr>	Existing attribute name	Name of the attribute that holds the source coordinate system string.	No
CONVERSION_DIRECTION	FROM_FME_TO or TO_FME_FROM	Indicates which way the conversion is to be done.	No
EXTERNAL_SYSTEM_NAME	AUTODESK, EPSG, ESRI, MAPINFO or OGC	Indicates the external system name.	No
<outputCoordsysAttr>	Existing attribute name	Name of the attribute which will hold the output coordinate system string	No

Configuration

The @CoordSys function does not accept configuration lines.

Description

This function returns the name of the coordinate system where this feature's coordinates are measured. If it returns a blank, it means that no coordinate system has been set for the feature data. If a coordinate system definition is created automatically by FME from parameters read from an input data source, its name will begin with an underscore character (_).

FME may automatically define coordinate systems. It is possible that there may be two different names for the same coordinate system definition. The result returned by @CoordSys should not be used in a comparison with a hardcoded coordinate system name as coordinate system names may be different but may be structurally equivalent.

Inverse Operation

The function performs the same operation in the inverse direction.

Example

In this example, the feature coordinate system name is stored within the attribute named `coordSys`.

```
SHAPE lake
MIF lake coordSys @CoordSys()
```

@CopyAttributes

```
@CopyAttributes(<targetAttrName>,<sourceAttrName> \
               [,<targetAttrName>,<sourceAttrName>]+)
@CopyAttributes(<targetListName>{},<sourceListName>{})
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<targetAttrName>	String	The name of the attribute that will be created in the feature.	No
<sourceAttrName>	String	The name of the attribute that will supply the value for the created attribute.	No
<targetListName>	String	The name of a list attribute that will be created in the feature.	No
<sourceListName>	String	The name of the list attribute that will supply the values for the created list attribute.	No

Configuration

The @CopyAttributes function does not accept configuration lines.

Description

This function copies the value of the source attribute into the target attribute. If many copies are being done with one invocation, they are executed as a single atomic copy. In this way, the original value of all the source attribute names is used to supply the value for the target attributes, so an attribute value swap may be performed.

It may also be used to copy list structures from one list name to another on a feature. All parts of the list are copied, including any nested attributes.

Inverse Operation

This function does nothing when invoked in the reverse direction, which happens when it appears on the source portion of a transfer specification.

Example 1

In this example, @CopyAttributes is used to swap the values of attributes a and b in the feature.

```
FACTORY_DEF * SamplingFactory \
SAMPLE_RATE 1 \
INPUT FEATURE_TYPE * \
    @CopyAttributes(a,b,b,a)
```

Example 2

In this example, @CopyAttributes is used to copy the list text_elements{} to the list parts{} in the feature.

```
FACTORY_DEF * SamplingFactory \
SAMPLE_RATE 1 \
INPUT FEATURE_TYPE * \
    @CopyAttributes(parts{},text_elements{})
```

If the original feature had attributes like:

```
text_elements{0}      val0
text_elements{0}.size 30
text_elements{1}      val1
text_elements{1}.size 35
```

then after the function runs, these additional attributes will be present:

```
parts{0}      val0
parts{0}.size 30
parts{1}      val1
parts{1}.size 35
```

@Count

```
@Count ( [ <domain> [ , <startVal> [ , <modulo> ] ] [ , NO_LOG ] )
```

Function Type: Attribute or Feature

Arguments:

Name	Range	Description	Optional
<domain>	Any String	A counter name. Each time @Count is invoked, it returns and increments the count associated with the domain name. This allows many different counters to be used during a single translation. If this parameter is not specified, the default domain is assumed.	Yes
<startVal>	Any Integer	The starting value of the counter. The counter is incremented from the start value.	Yes
<modulo>	Any Integer	The modulo value of the counter. The counter returns a value between zero and <modulo> - 1.	Yes
NO_LOG		Prevents FME from logging this domain.	Yes

Configuration

The @Count function accepts the following configuration line:

```
Count MAX_TO_LOG <number>
```

If this configuration line is not present, a maximum of 50 count domains will be logged by default.

Name	Range	Description	Optional
<number>	Integer >= -1	Sets the maximum number of count domains that will be logged. If -1 is specified, all count domains are logged.	No

Description

This function provides a mechanism for generating unique numbers and assigning them to feature attributes during a translation. Because it outputs the final counts in each of the domains to the log file, it can also be used as a feature function to count features that matched the correlation lines. In this case, the log file records the total number of times the function was invoked, even though its result was not stored in any attribute.

The optional `domain` parameter is used to have several different counters active during a single translation. For example, unique line numbers starting at 0 can be assigned to all lines by invoking `@Count(lineCounter)`. During the same run, unique polygon numbers starting at 0 can be assigned to all polygons by using `@Count(polygonCounter)`.

If no domain name is specified, then the default domain will be assumed.

The optional `startVal` parameter specifies a starting value for the counter. This is useful for applications where ranges of values have meanings in the problem domain.

The optional `modulo` parameter enables a counter to be created that cycles from 0 to `modulo - 1`. This is useful when using counters as lookup values with the `@Lookup` function.

The `NO_LOG` flag may be specified to prevent the FME from logging this domain.

Inverse Operation

This function works identically in both forward and reverse directions.

Example

In the example below, when translating from a Design file to a Shape file, each line output to the Shape file has a unique number assigned to its `LINENUM` attribute. At the end of the run, the log file contains the number of times `@Count()` was invoked. Notice in this case that the default count domain was used.

```
SHAPE lines LINENUM @Count()
IGDS 42 igds_type igds_line
```


@CRC

```
@CRC(<directive> [,<optional attr list>]
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
@CRC (ALL)	String	Calculates CRC for all attributes and geometry and returns the CRC value as a string.	No
@CRC (GEOM)	String	Calculates the CRC for only the geometry.	Yes
@CRC (GEOM, <attr1>, ..., <attrN>)	String	Calculates the CRC for the geometry and the specified attributes.	Yes
@CRC (ATTRIBUTES)	String	Calculates the CRC for all attributes.	Yes
@CRC (ATTRIBUTES, <attr1>, ..., <attrN>)	String	Calculates the CRC for all attributes.	Yes

Configuration

The @CRC function does not accept configuration lines.

Description

This function is used to calculate a CRC value for geometry, selected attributes, or both as described above.

The CRC is returned as a string representing a 64 bit integer if the crc was calculated for both attributes and geometry, and a 32 bit integer if the crc was calculated for only geometry or only attribution.

Inverse Operation

This function has no inverse.

Example 1

In the example below, the function is being used to calculate a CRC value for the geometry and the specified attributes. The value is being put into an attribute called `crc`.

```

FACTORY_DEF * TeeFactory \
    INPUT FEATURE_TYPE * \
    OUTPUT FEATURE_TYPE * \
    crc @CRC(GEOM,ALT1_NAME,FCC,HWYNAME,LENGTH,STATEFIPS)

```

Example 2

In the example below, the function is being used to calculate a CRC value for only the specified attributes. The value is being put into an attribute called `crc`.

```

FACTORY_DEF * TeeFactory \
    INPUT FEATURE_TYPE * \
    OUTPUT FEATURE_TYPE * \
    crc @CRC(ATTRIBUTES,ALT1_NAME,FCC,HWYNAME,LENGTH,STATEFIPS)

```

Example 3

In the example below, the function is being used to calculate a CRC value for only the geometry. The value is stored in an attribute named `crc`.

```

FACTORY_DEF * TeeFactory \
    INPUT FEATURE_TYPE * \
    OUTPUT FEATURE_TYPE * \
    crc @CRC(GEOM)

```

| @CSG

| @CSG ()

Function Type: Feature

Arguments: None

Configuration

The @CSG function does not accept configuration lines.

Description

This function is used to replace the CSG solid geometry of a feature by evaluating the CSG solid. The result could be a multi-solid, a BRep solid, or a NULL geometry.

Features that do not have CSG geometry are untouched.

Inverse Operation

This function has no inverse.

@Dimension

```
@Dimension([(2 | 3)])
```

Function Type: Attribute *or* Feature

Arguments: None

Configuration

The @Dimension function does not accept configuration lines.

Description

If this function does not have any parameters, it will return the dimension of the feature. In the current version of FME, 2 or 3 will be returned.

If this function is given 2 or 3 as a parameter, it will force the feature to that dimension and will return the new dimension of the feature.

Inverse Operation

The function performs the same operation in the inverse direction.

Example

In the example below, the feature dimension is stored within the attribute named dimension.

```
SHAPE lake
MIF lake dimension @Dimension()
```

@Evaluate

@Evaluate(<expression>)

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<expression>	Any Valid Expression	The expression may contain any of these operators: feature variable references (using the value-of operator &); transfer variables; or calls to other functions.	No

Configuration

The @Evaluate function does not accept configuration lines.

Description

This function evaluates an arithmetical expression and returns the result. The operators permitted in the expressions to be evaluated are a subset of the operators permitted in c expressions. They have the same meaning and precedence as the corresponding c operators.

Tip

If the expression contains spaces or nested parentheses, it should be placed in quotes.

Expressions frequently yield numeric results, such as integer or floating-point values. For example, the expression:

```
@Evaluate("8.2 + 6")
```

returns 14.2.

@Evaluate is based on the Tool Command Language (TCL, version 8.4.12) `expr` command, as is the documentation below. TCL and its documentation is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., and other parties. However, the TCL authors have granted permission to any party to reuse and modify the code and documentation, provided the original copyright holders are acknowledged.

Operands

An expression consists of a combination of operands, operators, and parentheses. White space may be used between the operands, operators, and parentheses as it is ignored by the expression processor. Where possible, operands are interpreted as integer values. Integer values may be specified in decimal which is the normal case, in octal if the first character of the operand is 0, or in hexadecimal if the first two characters of the operand are 0x. If an operand does not have one of the integer formats given above, then it will be treated as a floating-point number if possible. Floating-point numbers may be specified in any of the ways accepted by an ANSI-compliant C compiler, except that "f", "F", "l", and "L" suffixes are not permitted in most installations. For example, all of the following are valid floating-point numbers: 2.1, 3., 6e4, 7.91e+16. If no numeric interpretation is possible, then an operand is left as a string and only a limited set of operators may be applied to it.

Operands may be specified in any of the following ways:

- As a numeric value, either integer or floating-point.
- As a value of an attribute, using standard & notation. The attribute's value is used as the operand.
- As an FME feature function, such as `@Area()`. The function is evaluated and the result used as the operand.
- As a mathematical function whose arguments have any of the above forms for operands, such as `sin(&x)`. See the table below for a list of defined mathematical functions.

Operators

The valid operators listed below are grouped in decreasing order of precedence:

Operator	Description
- + ~ !	Unary minus, unary plus, bit-wise NOT, logical NOT. None of these operands may be applied to string operands, and bit-wise NOT may be applied only to integers.
* / %	Multiply, divide, remainder. None of these operands may be applied to string operands, and remainder may be applied only to integers. The remainder always has the same sign as the divisor and an absolute value smaller than the divisor.
+ -	Add and subtract. Valid for any numeric operands.
<< >>	Left and right shift. Valid for integer operands only.

Operator	Description
< > <= >=	Boolean less, greater, less than or equal, and greater than or equal. Each operator produces 1 if the condition is true, 0 otherwise. These operators may be applied to strings as well as numeric operands, in which case string comparison is used.
== !=	Boolean equal and not equal. Each operator produces a zero/one result. Valid for all operand types.
&	Bit-wise AND. Valid for integer operands only.
^	Bit-wise exclusive OR. Valid for integer operands only.
	Bit-wise OR. Valid for integer operands only.
&&	Logical AND. Produces a 1 result if both operands are non-zero, 0 otherwise. Valid for numeric operands only (integers or floating-point).
	Logical OR. Produces a 0 result if both operands are zero, 1 otherwise. Valid for numeric operands only (integers or floating-point).
x?y:z	If-then-else, as in C. If x evaluates to non-zero, then the result is the value of y. Otherwise, the result is the value of z. The x operand must have a numeric value.
- + ~ !	Unary minus, unary plus, bit-wise NOT, logical NOT. None of these operands may be applied to string operands, and bit-wise NOT may be applied only to integers.
* / %	Multiply, divide, remainder. None of these operands may be applied to string operands and remainder may be applied only to integers. The remainder always has the same sign as the divisor and an absolute value smaller than the divisor.

See the *C Manual* for more details on the results produced by each operator. All binary operators group left-to-right within the same precedence level. For example, the expression:

```
@Evaluate(4*2 < 7)
```

returns 0.

Math Functions

@Evaluate supports the following mathematical functions in expressions:

acos	cos	hypot	sinh	asin
cosh	log	sqrt	atan	exp
log10	tan	atan2	floor	pow
tanh	ceil	fmod	sin	

Each of these functions invokes the C math library function of the same name. Refer to the *C Manual* entries for the library functions for details on what they do. @Evaluate also implements the following functions for conversion between integers and floating-point numbers:

Function	Description
<code>abs(arg)</code>	Returns the absolute value of <code>arg</code> . <code>arg</code> may be either integer or floating-point, and the result is returned in the same form.
<code>double(arg)</code>	If <code>arg</code> is a floating value, returns <code>arg</code> . Otherwise converts <code>arg</code> to floating point and returns the converted value.
<code>int(arg)</code>	If <code>arg</code> is an integer value, returns <code>arg</code> . Otherwise converts <code>arg</code> to integer by truncation and returns the converted value.
<code>round(arg)</code>	If <code>arg</code> is an integer value, returns <code>arg</code> . Otherwise converts <code>arg</code> to integer by rounding and returns the converted value.

Types, Overflows, Precision

All internal computations involving integers are done with C-type long, and all internal computations involving floating-point are done with C-type double. When converting a string to floating-point, exponent overflow is detected and results in an error. For conversion to integer from string, detection of overflow depends on the behaviour of some routines in the local C library, so it should be regarded as unreliable. In any case, integer overflow and underflow are generally not reliably detected for intermediate results. Floating-point overflow and underflow are detected to the degree supported by the hardware, which is generally reliable.

Conversion among internal representations for integer, floating-point, and string operands is automatically done as needed. For arithmetical computations, integers are used until some floating-point number is introduced, after which floating-point is used. For example,

```
@Evaluate("5 / 4")
```

returns 1, while

```
@Evaluate("5 / 4.0")
@Evaluate("5 / (4 + 0.0)")
```

both return 1.25. Floating-point values are always returned with a "." or an "e" so that they will not look like integer values. For example,

```
@Evaluate(20.0/5.0)
```

returns "4.0", not "4".

Seventeen digits of precision are always used for floating point calculations.

Inverse Operation

This function has no inverse.

Example

In the example below, @Evaluate is used to compute the ratio of the number of rooms in a building to the number of stories. The result is used in a test by the TestFactory to route the features to two different destinations based on the density of rooms per floor in the building.

```
FACTORY_DEF SHAPE TestFactory \
  INPUT FEATURE_TYPE Building \
    TEST @Evaluate(&numRooms/&stories) < 10 \
    OUTPUT PASSED FEATURE_TYPE SparseBuilding \
    OUTPUT FAILED FEATURE_TYPE DenseBuilding
```

@FeatureType

@FeatureType ([<newType>])

Function Type: Attribute *or* Feature

Arguments:

Name	Range	Description	Optional
<newType>	String	A new feature type for the feature.	Yes

Configuration

The @FeatureType function does not accept configuration lines.

Description

This function is used as either an attribute value function or a feature function. When used as a feature function, the optional `newType` parameter must be specified. In this case, the @FeatureType function changes the feature type of the feature to the specified value.

When used as an attribute value function, the `newType` parameter is not specified. In this case, @FeatureType returns the feature type of the feature. This value is then stored in an attribute.

This function is primarily used in conjunction with factories or the wildcard feature type. On factory input lines, it may be used to store the feature type in an attribute for use in a group-by clause. On a factory output line, the function may be used to set the feature type of an output feature from one of its attributes.

When used with the wildcard feature type in a transfer specification, this function acts as a feature function to either set the feature type when invoked in the forward direction, or to set a transfer variable to the type of the feature when invoked in the reverse direction.

Tip

The wildcard feature type matches all feature types and is denoted by an asterisk.

Inverse Operation

When encountered on the source line of a transfer specification, @FeatureType does nothing if the optional newType parameter is **not** specified using a transfer variable.

However, when @FeatureType is called with a transfer variable on a source line of a transfer specification, it sets the transfer variable to the feature type of the feature.

Example

The example below shows how the wildcard feature type and the @FeatureType function can be used to provide generic, no-value-added translation between Design and Shape files. In this example, all linear features, regardless of their original Design feature type (level), get stored in the same Shape file. When the translation goes from Design to Shape files, the @FeatureType(%level) is run in the inverse direction, assigning the feature type of the current feature to the %level variable. This variable is then assigned to the LEVEL field of the output Shape linear feature.

When translation goes from a Shape to a Design file, the @FeatureType(%level) runs in the forward direction, and assigns the value in the %level variable to the feature type of the current feature. In the case of Design files, the feature type is interpreted as the level when the feature is output.

```
SHAPE igdsline LEVEL %level COLOR %color STYLE %style
IGDS *  igds_type igds_line igds_color %color          \
      igds_style %style                                \
      @FeatureType(%level)
```

@File

@File(<oper>, <attributeName>, <fileName>)

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<oper>	DestWriteSrcRead or DestAppendSrcRead or DestReadSrcWrite or DestReadSrcAppend	<p>DestWriteSrcRead — On the destination side of a correlation line pair or on factory lines, the contents of the attribute are written to the file. If the file exists, then any existing value is overridden.</p> <p>On the source line of a correlation line pair, the data is read from the file and stored in the attribute.</p> <p>DestAppendSrcRead — On the destination side of a correlation line pair or on factory lines, the contents of the attribute are appended to the file. If the file exists then the information is simply written to the end of the file. On the source line of a correlation line pair, the data is read from the file and stored in the attribute.</p> <p>DestReadSrcWrite — On the source side of a correlation line pair, the contents of the attribute are written to the file. If the file exists, then any existing value is overridden. On the destination side of a correlation line pair or on a factory line, the data is read from the file and stored in the attribute.</p> <p>DestReadSrcAppend — On the source side of a correlation line pair the contents of the attribute are appended to the file. If the file exists, then any existing value is overridden. On the destination side of a correlation line pair or on a factory line, the data is read from the file and stored in the attribute.</p>	No

Name	Range	Description	Optional
<attributeName>	String	The name of the attribute whose value is to be written to the file or read from the file. The entire contents of the file are stored in this attribute of the feature.	No
<fileName>	String	The name of the target file from which data is read or to which data is written.	No

Configuration

File <fileName> <directive>

Name	Range	Description	Optional
<fileName>	String	The name of the target file.	No
<directive>	STARTUP_DELETE SHUTDOWN_DELETE	STARTUP_DELETE — The specified file <fileName> is deleted from the system during FME start-up to ensure that the file is empty when the first @File function is executed. SHUTDOWN_DELETE — The specified file <fileName> is deleted from the system during the FME shutdown process to ensure that the file is not left laying around after the translation is complete.	No

Description

This function enables FME to read the contents of a file into a feature attribute or write the contents of a feature attribute out to a file. The configuration lines are optional, in which case the files are deleted during FME start-up or shutdown as directed by the configuration line.

Inverse Operation

Performs operation as specified by the `src` portion of the description. See the preceding paragraph for the description.

Example

The following example combines @System() and @File. @System is used to compress a file, then @File is used to read the zip file into an attribute named DRAWING_DATA. This is particularly useful when the destination system is a database supporting blob-type columns, because then it is possible for the database to store any type of associated data.

```
FACTORY_DEF * SamplingFactory                                \  
    SAMPLE_RATE 1                                           \  
    INPUT FEATURE_TYPE *                                    \  
        @System("zip $(destFILE) c:\tmp\descript.doc")      \  
        @File(DestReadSrcWrite,DRAWING_DATA,$(destFILE))
```


@Filenamepart

@Filenamepart (<keyword>,<filename>)

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<keyword>	DIRNAME FILENAME ROOTNAME EXTENSION	The part of the filename you want ex- tracted.	No
<filename>	Any String	The full pathname of a file.	No

Configuration

The @Filenamepart function does not accept configuration lines.

Description

This function extracts a part of a full file name it is passed in and returns the result as a string.

When parsing the full file name, both backslashes and forward slashes are handled as path separators, regardless of operating system.

For example, the full path name

C:\WINNT\Profiles\user\Desktop\roads.shp

has a DIRNAME of

C:\WINNT\Profiles\user\Desktop\

and a FILENAME of

"roads.shp",

and a ROOTNAME of

"roads",

and an EXTENSION of

"shp".

Inverse Operation

This function works identically in both forward and reverse directions.

Example

In the example below, the feature type is set to the rootname of the file from which it was read.

```
FACTORY_DEF DWG SamplingFactory \  
  SAMPLE_RATE 1 \  
  INPUT FEATURE_TYPE *\  
    @FeatureType(@Filenamepart(ROOTNAME,$(SourceDataset)))
```

@FitDGN

Note: This function is not supported by FME Base Edition.

```
@FitDGN(<fileName>, <designUnit>, <buffer>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<fileName>	String	The Design file to be fit into the bounding box of the feature upon which it is invoked.	No
<designUnit>	MASTER SUBUNIT	MASTER – The file is fit into the bounding box of the passed-in feature using master units. SUBUNIT – The file is fit into the bounding box of the passed-in feature using subunits.	No
<buffer>	Real	The buffer percentage. This defines the percentage of white space to leave around the edge of the bounding box when fitting the Design file.	No

Configuration

This function does not accept configuration lines.

Description

This function moves and scales the specified Design file to fit within the bounding box of the feature on which it is called. The amount of area to leave empty as border around the outside of the moved and scaled Design file is specified using <buffer>. <buffer> is a percentage and must fall in the range 0 <= <buffer> < 100. The MASTER or SUBUNIT flag indicates how the units in the design file should be interpreted, compared to the units of the feature that the function is invoked upon.

Note: This function does not create a copy of the specified Design file, but rather directly modifies the Design file's global origin and scaling factors so it fits into the bounding box of the feature.

Inverse Operation

This function does nothing when performed in the inverse direction.

Example

The following example combines @FitDGN(), @System(), and @File().

@FitDGN() is used to move the specified Design file to the correct location. @System() is used to compress the moved file, then @File() is used to read the zip file into an attribute named DRAWING_DATA. This is particularly useful when the destination system is a database supporting blob-type columns because it is then possible for the database to store any type of associated data.

```

FACTORY_DEF * SamplingFactory                                \
                                                                    \
SAMPLE_RATE 1                                                \
INPUT FEATURE_TYPE *                                         \
    @FitDGN(c:\tmp\drawing.dgn, MASTER, 5)                  \
    @System("zip $(destFILE) c:\tmp\drawing.dgn")           \
    @File(DestReadSrcWrite,DRAWING_DATA,$(destFILE))

```

@Force2D

@Force2D()

Function Type: Feature

Arguments: None

Configuration

The @Force2D function does not accept configuration lines.

Description

This function forces the feature to be 2D and returns the original dimensionality of the feature. If the input feature was 3D, then the z dimension will be dropped. If the input feature is 2D, then the feature will be untouched.

Tip

@Force2D is useful when outputting data to a format like AutoCAD files, since many AutoCAD-compatible products cannot process 3D files.

Inverse Operation

The function performs the same operation in the inverse direction.

Example

In the example below, a SamplingFactory is used with the @Force2D function to convert all input features to 2D. The SamplingFactory uses a sample rate of 1, meaning every feature will pass through it.

```
FACTORY_DEF IGDS SamplingFactory          \
SAMPLE_RATE 1                             \
INPUT FEATURE_TYPE * @Force2D()
```

@Generalize

```
@Generalize(Deveau,<tolerance>,<numWedges>,<angle>)
@Generalize(Douglas,<tolerance>)
@Generalize(Thin,<tolerance>)
@Generalize(ThinNoPoint,<tolerance>)
@Generalize(McMaster,<numNeighbors>,<displacement>)
@Generalize(McMasterWeightedDistance,<numNeighbors>,<displacement>,<weightingPower>)
@Generalize(Curvefit,<precision>,<flattening>,<compressionWgt>,<smoothnessWgt>,<accuracyWgt>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<tolerance>	>= 0	<p>The tolerance used for point thinning or generalization. For the Douglas algorithm, points less than the tolerance from the surrounding line segment are removed. The result is that all points of the original segment will be within a band of width <tolerance> <i>centered</i> around the resulting line.</p> <p>For the Deveau algorithm, points are kept only when no band of width <tolerance> can be found that contains both the original points and the resulting line. The band is allowed to <i>float</i>, resulting in a smoother result.</p> <p>For the Thin and ThinNoPoint algorithms, points are removed such that the distance between adjacent spaces is guaranteed to be larger than <tolerance>.</p> <p>When a value of 0 is given for the tolerance, only redundant nodes are removed. (“Redundant” nodes are defined this way: when a redundant node is removed, it does not change the shape of the line in any way.)</p>	No

Name	Range	Description	Optional
<numWedges>	1..30	For the Deveau algorithm, this controls the number of simultaneous wedges considered when forming floating bands around the points in the set. When <numWedges> is 1, the Deveau algorithm functions in the same way as the Douglas algorithm. The larger this value is, the more aggressive the generalization and the smoother the resulting line.	No
<angle>	Between 0 and 180	This parameter sets the <i>sharpness</i> tolerance for spikes that will be blunted. Vertex points at angles less than <angle> from the previous two points are not moved. The angle is measured in degrees. A value of 110 is recommended.	No
<numNeighbors>	>= 1	For the McMaster and McMasterWeightedDistance algorithms, each point is smoothed by taking an average of its x and y coordinates and the x and y coordinates of neighboring points. This parameter determines how many neighbors to the left and right of each point are considered.	No
<displacement>	Between 0 and 100	For the McMaster and McMasterWeightedDistance algorithms, after a point has been given a new location through averaging, it is then moved towards its original location by an amount specified by this parameter. Note that this parameter is a percentage, so a value of 50 will place the point directly in the middle of its averaged location and its original location.	No
<weightingPower>	>= 0	For the McMasterWeightedDistance algorithm, this parameter is used as the power in an inverse distance weighting formula. A value of zero will give each point in the averaging calculations equal weight. A value from 0 to 3 is recommended.	No
<precision>	> 0	For the Curvefit algorithm, this sets the maximum deviation allowed at any point along the polyline between the original and the resulting polyline.	
<flattening>	>= 0	For the Curvefit algorithm, this allows very flat curves to be represented by straight segments. Any curve that has a mid-ordinate less than this amount will be replaced by a straight segment. A typical value is 10% of the precision setting.	

Name	Range	Description	Optional
<compressionWgt>	> 0, <= 10	For the Curvefit algorithm, this determines the importance given to Compression relative to Smoothness and Accuracy. Compression is the reduction in the number of vertices	
<smoothnessWgt>	> 0, <= 10	For the Curvefit algorithm, this determines the importance given to Smoothness relative to Compression and Accuracy. Smoothness is the tangency of consecutive segments - how close the end angle of a segment is to the start angle of the next segment.	
<accuracyWgt>	> 0, <= 10	For the Curvefit algorithm, this determines the importance given to Accuracy relative to Compression and Smoothness. Accuracy is how closely the resulting curve overlays the original.	

Configuration

This function does not accept configuration lines.

Description

The @Generalize function modifies a feature's geometry by removing its points or by calculating new positions for its points.

Note There are two issues to consider before performing generalization. The first is that the feature should have no self-intersections. The second is that the generalization value should be less than any narrow corridors within the feature. For performance reasons, the @Generalize function does not ensure either of these conditions. If you think your data might have self-intersections, then you should first run it through the `IntersectionFactory` and then perform @Generalize on the resulting pieces. To detect and fix any narrow corridors that are less than the generalization value, you can use `IntersectionFactory` after @Generalize is performed.

Two algorithms are available for point thinning or generalizing: Douglas and Deveau.

If the `FME_GEOMETRY_HANDLING` directive is set to "yes" in the mapping file, arcs and ellipses are not touched, while the location of text features are generalized; otherwise, the result of generalizing arcs, ellipses, and text are single points located at their respective center points.

Douglas Algorithm

The Douglas algorithm takes only a `<tolerance>` parameter to the amount of point thinning performed. The Douglas algorithm removes points from the original line, but does not adjust the location of the remaining points.

The Douglas algorithm is described in the following publication:

David H. Douglas and Thomas K. Peucker, “*Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or Its Caricature*,”
CANADIAN CARTOGRAPHER, Vol. 10, No. 2,
December 1973, pp 112-122

Deveau Algorithm

The Deveau algorithm both removes points and adjusts the locations of the remaining points, resulting in a smoother generalization. In addition to the `<tolerance>` parameter, the Deveau algorithm also takes `<numWedges>` and `<angle>` parameters to control the generalization. These parameters must be tuned for particular applications to produce aesthetically pleasing results. The Deveau algorithm is fully described in:

AutoCarto VII proceedings in the paper “*Reducing the Number of Points in a Plane Curve Representation*,” by
Terry J. Deveau.

Note The generalized output feature’s geometry will always be 2D, because of the way algorithm works. During generalization, existing vertices may be moved and/or new vertices introduced, which invalidates the z coordinate for the vertices.

Thin Algorithm

The Thin algorithm is a simple vertex thinning algorithm that only removes points – it does not adjust the locations of the remaining points. Points are removed such that the distance between adjacent vertices is guaranteed to be larger than the `<tolerance>` parameter.

ThinNoPoint Algorithm

The ThinNoPoint algorithm is the same as the Thin algorithm, except that the beginning and end points of lines are never moved. If the entire length of the feature being thinned is less than the tolerance, then the feature is replaced by a linear feature connecting the first point to the last point.

McMaster Algorithm

The McMaster algorithm smooths a feature by determining a new location for each of its points. The McMaster algorithm works by determining a new location for a point by taking the average of its x and y locations and the x and y locations of neighboring points. It then slides the point towards its original location. The overall effect is that each point is pulled towards its neighbors.

McMasterWeightedDistance Algorithm

The McMasterWeightedDistance algorithm works the same as the McMaster algorithm only it uses an inverse distance weighting formula to take into account the distance from a point to its neighboring points. Points that are close will have more pull on each other than points that are further apart.

Curvefit Algorithm

The Curvefit algorithm works by removing vertices and replacing the line segments between them with arcs. The process can be customized to emphasize vertex reduction, absolute fit, or tangency (smoothness) between segments.

Tip

The @Generalize function logs statistics about the number of points it removed using each algorithm.

Note that as the values of the weight settings increase, the number of potential solutions considered increases in steps. If any weight setting is 2 or greater, an increased number of potential solutions is computed; if any weight setting is 5 or greater, the maximum number of potential solutions is calculated. This increases the probability of finding the optimum solution, but also increases computation time.

Inverse Operation

This function has no inverse and is ignored when on the source side of a transformation specification.

Example

In the example below, SAIF_{road} features are generalized before being output to a Shape file. The Douglas algorithm is used to do this generalization.

```
SAIF Road::TRIM numberOfLanes %num paved %pavedFlag
SHAPE road NUMLANES %num PAVED %pavedFlag \
    @Generalize(Douglas,50)
```

@GeneratePoint

```
@GeneratePoint ( [ <xAttrName> , <yAttrName> ] )
```

Function Type: Feature

Name	Range	Description	Optional
<xAttrName>	Attribute Name	The name of the attribute that will be set to the the <i>x</i> coordinate value of the generated point.	Yes
<yAttrName>	Attribute Name	The name of the attribute that will be set to the the <i>y</i> coordinate value of the generated point.	Yes

Configuration

The @GeneratePoint function does not accept configuration lines.

Description

This function generates a point inside a polygon, or donut polygon, feature. The generated point is added to the feature's in-memory representation. If no attribute names are provided and the function is called with no parameters, the feature then becomes a point-in-polygon (PIP) feature. When generating a point for a 3D feature, the *z* coordinate is set to the average *z* value of the input feature.

If attribute names are provided, the feature's geometry does not change and instead the *x* and *y* coordinates of the generated point are added as attributes to the feature.

This function can be used when translating data from a model where polygons are directly attached, to a model where polygonal areas are implied by boundary lines, and the attribution for each area is attached to a point within that area.

If the input feature is not a polygon or donut polygon, it is returned by the function unchanged.

If the `FME_GEOMETRY_HANDLING` directive is set to "yes" in the mapping file, ellipses are regarded as polygons; they are otherwise regarded as points.

Inverse Operation

This function has no inverse.

Example

In the example below, polygonal data with attached attribution is read from a dataset, but the attribution is to be attached to point features output to a different dataset. To perform the split, the polygons are input to the

PIPComponentsFactory, and, as they are input, @GeneratePoint is used to attach a point to them. This generated point is then output by itself, along with all the attributes, by the OUTPUT POINT clause of the factory. Notice, in this case, that since no OUTPUT POLYGON clause is present, the polygons themselves are discarded.

```
FACTORY_DEF * PIPComponentsFactory \
  INPUT      FEATURE_TYPE polys @GeneratePoint() \
  OUTPUT POINT FEATURE_TYPE points
```

In the second example, area features receive two new attributes that hold the *x* and *y* coordinates of an internal point.

```
FACTORY_DEF * TeeFactory \
  INPUT  FEATURE_TYPE polys @GeneratePoint(pointX,pointY) \
  OUTPUT FEATURE_TYPE *
```

@GenerateRasterPalettes

@GenerateRasterPalettes(<palette key interpretation>, <max num palette entries>)

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<palette key interpretation>	UINT8 UINT16 UINT32	The desired key interpretation for the new palette, including data type and bit depth. UINT8 has at most 256 palette entries. UINT16 has at most 65,536 palette entries. UINT32 has at most 4,294,967,296 palette entries.	No
<max num palette entries>	positive integers	The actual number of entries in the new palette. The palette key interpretation may allow more entries in the palette, but this number sets the maximum number of entries to be generated. The minimum number is 1. This number cannot exceed the maximum number of entries in the palette key interpretation.	No

Configuration

This function does not accept configuration lines.

Description

The @GenerateRasterPalettes function is used to generate a palette from a raster without a palette. This function removes the selected band(s) from a raster and produces a new band and a new palette. The function replaces the values on the selected band(s) by the palette keys on the new band and the values on the new palette. This process often greatly reduce the size of the output raster. This also implies that the output raster has only as many different colors as the new palette. When the input raster has more different colors than the specified number of colors in the function parameter, the color reduction mechanism in the function will merge similar colors to make sure that the maximum number of colors in the palette will not exceed the specified number in the parameter.

Since this function can be used to generate a palette from a raster without a palette, translations from raster formats that do not produce a palette, to raster formats that require a palette are possible now. For example, when a raster writer, such as the GIFRASTER and the PNGRASTER, requires a palette, this function can be used to generate a palette from the source raster, such as the JPEG and the ECW, which does not have a palette.

In details, the selected band(s) of the input raster will be replaced by a new band with the specified palette key interpretation. This new band will have a new palette. The interpretations of the selected band(s) determine the value interpretation of the palette. For example, if the interpretations of the selected bands are RED8, GREEN8, and BLUE8, the value interpretation of the new palette will be RGB24. The specified maximum number of palette entries determines the maximum number of palette entries to be generated in the new palette. This number is useful in some situations. For example, XPM can write at most 87 entries in a palette, whereas GeoTIFF can write at most 2,147,483,648 entries. To standardize the number of entries and the color in the palettes being written to both formats, the @GenerateRasterPalettes function can be used by setting the key interpretation to UINT8 and the maximum number of palette entries to 87 before passing the raster feature to the writers of both formats.

This operation is lossy. The maximum number of entries determines the quality of the output raster. More number of entries implies higher quality.

The two arguments allow the function to generate a palette that can contain more entries than the actual number of entries to be generated. However, the specified maximum number of entries must not be larger than the number of entries of the specified key interpretation.

The selected band(s) determines the value interpretation of the palette. Therefore, the selected band(s) must be in the correct order in the raster, and the interpretations of the selected band(s) must be a meaningful palette value interpretation that FME currently supports. A bad example would be that the selected bands are RED8, GREEN16, and BLUE8 in order. Another bad example would be that the selected bands are GREEN8, BLUE8, and RED8 in order. Furthermore, the selected band(s) cannot have any palettes. Please note that if the selected bands are all UINT8 but the intended palette value interpretation is RGB24, the selected bands must be reinterpreted first.

This function supports sub-selection of raster bands.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a use of the @GenerateRasterPalettes function. In this example, the SamplingFactory is used to pass through every feature and those that have selected band(s) will be used to generate a new palette with at most 16 UINT8 keys per raster by the @GenerateRasterPalettes function.

```
FACTORY_DEF * SamplingFactory \
    SAMPLE_RATE 1 \
    INPUT FEATURE_TYPE * \
        @GenerateRasterPalettes(UINT8, 16) \
    OUTPUT FEATURE_TYPE *
```


@Geometry

```

@Geometry(FROM_ENCODED_STRING, <encodedXMLString>)
@Geometry(FROM_ATTRIBUTE, <attrName>)
@Geometry(FROM_ATTRIBUTE_BINARY, <attrName>)
@Geometry(FROM_ATTRIBUTE_BINARY_HEX, <attrName>)

@Geometry(TO_ATTRIBUTE, <newAttrName>)
@Geometry(TO_ATTRIBUTE_BINARY, <newAttrName>)
@Geometry(TO_ATTRIBUTE_BINARY_HEX, <newAttrName>)

@Geometry(SET_PROPERTIES, ELLIPSE, [<primRadius>],
    [<secRadius>], [<rotation>],
    [<orientation>], [<centerX>], [<centerY>],
    [<centerZ>])
@Geometry(SET_PROPERTIES, ARC, [<primRadius>],
    [<secRadius>], [<rotation>], [<startAngle>],
    [<sweepAngle>], [<startX>], [<startY>], [<startZ>],
    [<endX>], [<endY>], [<endZ>], [<centerX>],
    [<centerY>], [<centerZ>])
@Geometry(SET_PROPERTIES, TEXT,
    [<textString>],
    [<textSize>], [<textRotation>])

@Geometry(GET_PROPERTIES, ELLIPSE,
    [<newPrimRadiusAttr>], [<newSecRadiusAttr>],
    [<newRotationAttr>],
    [<newOrientationAttr>],
    [<newCenterXAttr>], [<newCenterYAttr>],
    [<newCenterZAttr>])
@Geometry(GET_PROPERTIES, ARC,
    [<newPrimRadiusAttr>],
    [<newSecRadiusAttr>], [<newRotationAttr>],
    [<newStartAngleAttr>],
    [<newSweepAngleAttr>], [<newStartXAttr>],
    [<newStartYAttr>], [<newStartZAttr>],
    [<newEndXAttr>], [<newEndYAttr>], [<newEndZAttr>],
    [<newCenterXAttr>], [<newCenterYAttr>],
    [<newCenterZAttr>])
@Geometry(GET_PROPERTIES, TEXT,
    [<newTextStringAttr>],
    [<newTextSizeAttr>],
    [<newTextRotationAttr>])

@Geometry(SET_MEASURES, POINT, <value>, [<measureName>])
@Geometry(SET_MEASURES, VERTEX, <index>, <value>, [<measure-
Name>])
@Geometry(SET_MEASURES, LINE, <srcAttrList>{},
    [<measureName>])
@Geometry(SET_MEASURES, AREA, <srcAttrList>{},
    [<measureName>])
@Geometry(SET_MEASURES, ARC,
    <startPointVal>, <endPointVal>, [<measureName>])

```

```

@Geometry(SET_MEASURES, ARC3POINTS,
    <startPointVal>, <midPointVal>, <endPointVal>,
    [<measureName>])
@Geometry(GET_MEASURES, POINT, <newAttr>, [<measureName>])
@Geometry(GET_MEASURES, VERTEX, <index>, <value>, [<measure-
Name>])
@Geometry(GET_MEASURES, LINE,
    <newAttrList>{}, [<measureName>])
@Geometry(GET_MEASURES, AREA,
    <newAttrList>{}, [<measureName>])
@Geometry(GET_MEASURES, ARC, <newStartPointAttr>,
    <newEndPointAttr>, [<measureName>])
@Geometry(SET_MEASURES, ARC3POINTS,
    <newStartPointAttr>, <newMidPointAttr>,
    <newEndPointAttr>, [<measureName>])

@Geometry(ADD_TEXT, <textString>, [<textSizeAttr>],
    [<textRotation>])
@Geometry(REMOVE_TEXT)

@Geometry(PART_COUNT, (NONRECURSIVE|RECURSIVE))

@Geometry(REMOVE_DUPLICATE_COORDS |
    REMOVE_DUPLICATE_COORDS_2D)

@Geometry(CREATE_DONUT_BRIDGES)

```

Function Type: Feature OR Attribute

Arguments:

Described below.

Configuration

This function does not accept configuration lines.

Description

@Geometry provides a number of ways to retrieve properties of a feature's geometry, and to replace or modify the geometry.

The FROM_ modes replace the feature's entire geometry with the one specified. FROM_ENCODED_STRING is used by the Creator transformer, which supplies <encodedXMLString> in Parseable Encoded FME XML. FROM_ATTRIBUTE, FROM_ATTRIBUTE_BINARY, and FROM_ATTRIBUTE_BINARY_HEX expect <attrName> to specify an attribute containing the result of a previous @Geometry call using TO_ATTRIBUTE, TO_ATTRIBUTE_BINARY, or TO_ATTRIBUTE_BINARY_HEX, respectively.

The `TO_modes` create an attribute named by `<newAttrName>`, whose value will be a description of the feature's current geometry. `TO_ATTRIBUTE` creates the attribute in FME XML. `TO_ATTRIBUTE_BINARY` uses an internal binary representation of the geometry. This is generally the most efficient way to store and retrieve the geometry's description. The next most efficient storage method is to use `TO_ATTRIBUTE_BINARY_HEX`, which stores the internal binary representation, encoded as hexadecimal digits. This can be used if the resultant attribute value must be stored in a format which cannot handle binary data.

The `SET_PROPERTIES` and `GET_PROPERTIES` modes will change (or retrieve) the properties of geometries matching the given type (`ELLIPSE`, `ARC`, or `TEXT`). If the geometry is of a different type (e.g. the function argument is `ARC` and the geometry is a polygon), the feature will be left unmodified. In the `SET_modes`, each property value may be given as a constant, a value-of expression - `&attribute`, or an FME function call - `@Value(attr)`. In the `GET_modes`, each argument specifies the name of the attribute into which to store the respective value.

The `SET_MEASURES` and `GET_MEASURES` modes will supply (or retrieve) the measures (with the given name, or the default measures if no measure name is supplied) of geometries matching the given type. If the geometry is of a different type (e.g. the function argument is `LINE` and the geometry is a point), the feature will not be modified. The parameters are treated in a similar fashion as in the `_PROPERTIES` modes described above. If no `<measureName>` is given, the default measure will be set/retrieved. Note that when setting line measures, the values to be used must exist on the feature as an attribute list. If the list has less elements than the line has points, the remaining values will be set to NaN and a warning will be logged. A path geometry that is by itself or part of an area is not supported and a warning will be logged. Please use `DeaggregateFactory` to split the paths into its segments and supply (or retrieve) the measures on the individual segment. Afterwards, `ConnectionFactory` or `ArcFactory` can be used to join up the segments into path again.

The `_PROPERTIES` and `_MEASURES` modes all have optional parameters (indicated by square brackets). For any parameter omitted, the respective value will be unmodified (in a `SET_mode`), or will not be retrieved into an attribute (in a `GET_mode`).

`ADD_TEXT` will result in a text geometry having the feature's previous geometry as its location. `REMOVE_TEXT` will extract the text's location and set it as the feature's geometry. (Nothing will result from a `REMOVE_TEXT` call on a non-text geometry.)

The `PART_COUNT` option will cause `@Geometry` to return the number of parts in the geometry. For multis and aggregates, this is the number of parts, for paths, the number of segments, and for donuts, the number of shells (outer and inner); otherwise it is one. If `RECURSIVE` is given then aggregates' parts will also have their parts counted. Note that this recursion never causes the count to descend

into area boundaries which are paths - that is, a polygon is always considered one part, a donut's number of parts is always the number of boundaries, even if some of the boundaries are paths. In a multiarea, `RECURSIVE` does cause member donuts' shells to be counted.

In the `REMOVE_DUPLICATE_COORDS` and `REMOVE_DUPLICATE_COORDS_2D` modes, all elements of the geometry that are lines will be checked for duplicate coordinates. Any consecutive coordinates with the same location will be reduced to a single coordinate. Only X, Y, and possibly Z values are considered in comparing coordinates for equality—measures are ignored. (Z values are also ignored if the mode is `REMOVE_DUPLICATE_COORDS_2D`.)

The `CREATE_DONUT_BRIDGES` option is used to build connections between holes with the outer boundary in donut polygons. The result is a polygon-equivalent representation of the input donut. A single, connected path visits the boundary and each donut hole exactly once. This action is performed on all donuts contained in an input feature. The generated polygon boundary is guaranteed to be non self-intersecting if the input feature is properly oriented. Input features that have faulty donut topology may be rejected, and these features are marked with the additional attribute `_fme_donutbridge_failure_`.

Inverse Operation

The command has no effect in the inverse direction.

@GeometryTraits

```
@GeometryTraits(REMOVE_TRAITS, [<filter_exp>])

@GeometryTraits(SET_TRAITS|FETCH_TRAITS, [<regexp>],
<overWriteExisting>)

@GeometryTraits(FETCH_TRAITS, [<regexp>], <over-
WriteExisting>, <prefix>)

@GeometryTraits(SET_TRAITS_LIST, "<attr_list>",
<overWriteExisting>)

@GeometryTraits(FETCH_TRAITS_LIST, yes, "", [<trait-
name>]+)
```

Function Type: Feature or Geometry

Arguments:

Name	Range	Description	Optional
REMOVE_TRAITS	N/A	Designation to remove traits from geometry.	No
SET_TRAITS	N/A	Designation to copy attributes from the feature onto the geometry as traits.	No
FETCH_TRAITS	N/A	Designation to copy a geometry's traits onto a feature as attributes.	No
SET_TRAITS_LIST	N/A	Similar purpose as SET_TRAITS except that the input will include a list of attributes.	No
FETCH_TRAITS_LIST	N/A	Similar purpose as FETCH_TRAITS except that the input will include a list of traits.	No

Name	Range	Description	Optional
<filter_exp>	String	<p>In a function call, denotes the regular expression used to parse which geometry traits to remove.</p> <p>In a transformer usage, denotes the string expression to use to determine which geometry traits to remove.</p> <p>By default, if left NULL, all geometry traits will be removed.</p>	Yes
<regex>	String	Regular expression used to parse which traits/attributes to copy over to the feature/geometry. If not specified, all traits/attributes will be copied.	Yes
<overWriteExisting>	yes no	Determines whether to overwrite pre-existing traits/attributes when using the Set or Fetch functionality. Default is set to yes.	No
<prefix>	String	Prepended to the new attribute when it is copied over from the geometry by the Fetcher	Yes
<attr_list>	String	Space delimited list of attributes that are to be copied over to the geometry as traits.	No
<traitname>	String	Comma-delimited list of geometry traits that are to be copied over to the feature as attributes.	No

Configuration

This function does not accept configuration lines.

Description

Geometry traits are similar to attributes on a feature; they are defined as any kind of user-defined data that is stored onto a geometry. Previously data such as this could only be stored at the feature level, but now each separate geometry

can hold their own specific data. This capability is useful for situations where geometries are combined or split apart within features.

There are three ways to manipulate geometry traits via this function. The simplest allows the removal of traits from the geometry by either using a regular expression, or via a string expression that filters out the relevant traits to remove. By default all traits are removed unless otherwise specified. The other two uses of the function allow data to be copied between the feature and the geometry, using the Set and Fetch operations. During Set, feature attributes are copied over to the geometry as traits, and during Fetch the opposite occurs. By default all attributes or traits will be copied over unless otherwise specified by using a regular expression, a space-delimited list of attributes (Set), or a comma-delimited list of traits (Fetch). By design, the transformers that invoke this function will use regular expressions as input. As well, by default, the option to overwrite pre-existing traits/attributes is always set to True unless otherwise specified.

Inverse Operation

There is no inverse operation of the function.

Example

Removes all traits on a geometry:

```
FACTORY_DEF * TeeFactory
FACTORY_NAME GeometryTraitRemover
INPUT FEATURE_TYPE Surface
OUTPUT FEATURE_TYPE @GeometryTraits("REMOVE_TRAITS", "")
```

Remove traits using a filter:

```
FACTORY_DEF * TeeFactory
FACTORY_NAME GeometryTraitRemover
INPUT FEATURE_TYPE Surface
OUTPUT FEATURE_TYPE @GeometryTraits("REMOVE_TRAITS", "xml_")
```

Copy the attribute "Desc" onto the geometry as a trait, and do not overwrite any pre-existing traits of the same name:

```
FACTORY_DEF * TeeFactory
FACTORY_NAME GeometryTraitSetter
INPUT FEATURE_TYPE Surface_0
OUTPUT FEATURE_TYPE GeometryTraitSetter_OUTPUT
@GeometryTraits(SET_TRAITS_LIST, "Desc", no)
```

Copies the traits Width and Colour onto the feature as attributes:

```
FACTORY_DEF * TeeFactory
FACTORY_NAME GeometryTraitFetcher
INPUT  FEATURE_TYPE Surface_0
OUTPUT FEATURE_TYPE GeometryTraitFetcher_OUTPUT
@GeometryTraits(FETCH_TRAITS_LIST, "yes", "", Width, Colour)
```


@GeometryType

```
@GeometryType()
```

```
@GeometryType((fme_point|fme_line|fme_polygon))
```

```
@GeometryType((fme_line|  
fme_polygon),HANDLE_ARCS_AND_ELLIPSES)
```

```
@Geometry-  
Type(fme_polygon,CHECK_Z_IN_DUPLICATE_POINTS_REMOVAL  
)
```

```
@GeometryType(fme_arc,[rotation],<primaryRa-  
dius>,[secondaryRadius],<startAngle>,<sweepAngle>)
```

```
@GeometryType(fme_ellipse,<primaryRadius>,[second-  
aryRadius],[rotation],<orientation>)
```

```
@GeometryType(fme_text,<textString>,<text-  
Size>,[rotation])
```

```
@GeometryType(fme_face, <toleranceValue>)
```

```
@GeometryType(fme_extrusion, <extrusionX>, <extru-  
sionY>, <extrusionZ>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<geometryType>	fme_point fme_line fme_polygon fme_arc fme_ellipse fme_text	The fme_type being set on the feature.	No
HANDLE_ARCS_AND_ELLIPSES	N\A	This flag has the same effect as the 'FME_GEOMETRY_HANDLING' directive in the mapping file. See the 'Description' section for more information on the use of 'FME_GEOMETRY_HANDLING' directive.	N\A
<rotation>	Real value	For arcs and ellipses, this is the rotation of the primary axis, as measured counterclockwise from horizontal. For text, this is the rotation of the text string, also measured counterclockwise from horizontal.	Yes
<primaryRadius>	Real value	The radius of the ellipse or the arc along the primary axis.	No
<secondaryRadius>	Real value	The radius of the ellipse or the arc along the secondary axis.	Yes
<startAngle>	Real value	Refer to the @Arc (function) in the FME Functions and Factories manual for a detailed definition.	No
<sweepAngle>	Real value	Refer to the @Arc (function) in the FME Functions and Factories manual for a detailed definition.	No
<orientation>	LEFT_HAND_RULE RIGHT_HAND_RULE	LEFT_HAND_RULE specifies a counterclockwise arc direction. RIGHT_HAND_RULE specifies a clockwise arc direction.	No
<textString>	String	The text string.	No

Name	Range	Description	Optional
<textSize>	Real value	The text height.	No
CHECK_Z_IN_DUPLICATE_POINTS_REMOVAL	N/A	This flag, if present, will remove duplicate coordinates from a polygon only if the coordinates are equal in their X, Y and Z values. Otherwise, only X and Y values are considered.	N/A
<toleranceValue>	Real value	The tolerance used to check for the planarity of the donut or polygon.	No
<extrusionX>	Real Value	The X component of the extrusion vector	No
<extrusionY>	Real Value	The Y component of the extrusion vector	No
<extrusionZ>	Real Value	The Z component of the extrusion vector	No

Configuration

This function does not accept configuration lines.

Description

This command is responsible for returning or setting the geometry type of the feature upon which it is called.

If it is called with no parameters, it will return the geometry type of the feature. The geometry types that are returned will be one of `fme_point`, `fme_line`, `fme_polygon`, `fme_donut`, `fme_pip`, `fme_aggregate`, or `fme_undefined`. Search the *FME Universal Translator on-line help* for a description of these geometry types.

If the `<geometry type>` argument is specified, then the geometry type of the feature will be set to the specified type if the conditions discussed in the following paragraphs are met.

This function currently allows only simple geometry types to be set. It isn't possible to set a feature's geometry type to `fme_aggregate`, `fme_donut`, or `fme_pip`. See the `AggregateFactory` and `DonutFactory` for a description of how to construct features with these geometries.

It isn't possible to change the geometry type of a feature that is currently tagged as being one of `fme_aggregate`, `fme_donut` or `fme_pip`. See the `DeaggregateFactory`, `DonutHoleFactory`, and `PIPComponentsFactory` for a description of how to manipulate and change features with these geometry types.

If an attempt is made to set a feature's geometry type to `fme_point` and the feature has more than one coordinate, then the feature's geometry type is unchanged and no error will be reported by the function.

If an attempt is made to set a feature's geometry type to `fme_polygon` and the feature has more than one coordinate, then the first coordinate and the last coordinate must be the same or the feature's geometry type will be unchanged. The feature will also be cleaned up if there are duplicate points so that it leaves the method as a valid FME polygon.

If an attempt is made to set a feature's geometry type to `fme_line` or `fme_polygon` and the feature had only one coordinate, then the feature's geometry type is unchanged.

If an attempt is made to set a feature to `fme_arc` or `fme_ellipse`, the first coordinate will be used as the center point. An exception to this is if the input feature itself is an `fme_arc` or `fme_ellipse`, in which case the center point of the original geometry is carried over into the new geometry. If there aren't any coordinates, the center point will be created at a default location.

If an attempt is made to set a feature to `fme_text`, the existing geometry will be used as the text location. An exception to this is if the input feature is an `fme_arc` or `fme_ellipse`, in which case the center point of the original geometry is carried over into the new geometry. Another exception to this is when the existing geometry is a text string, in which case the new string's values will just replace the old values.

If the `FME_GEOMETRY_HANDLING` directive is set to "yes" in the mapping file, arcs and ellipses can have their geometry type set to `fme_point`, `fme_line` or `fme_polygon`; otherwise, arcs and ellipses will always be converted into points.

The `fme_face` mode attempts to convert the 3D polygon or donut geometry of a feature to a face geometry if it passes the planarity check. If the tolerance value is negative, the planarity check is skipped. If the tolerance value is greater than or equal to 0, the planarity check only passes if the polygon or donut is planar within the tolerance. If the polygon or donut is not closed in 3D, the first point and last point of the area are moved to a point that is an average of the two points.

The `fme_extrusion` mode converts a face geometry of a feature to a extrusion with the extrusion vector being `<extrusionX, extrusionY, extrusionZ>`.

Inverse Operation

The function always returns the geometry type of the feature when executed in the inverse direction (source side of correlation line pair).

Example

The following example shows how to use the @GeometryType function to downgrade features tagged as polygons to be treated as linear features.

The example below uses the TeeFactory to accept only features that are going to have their geometry changed to lines. Features have their geometry changed if they have an attribute named `lineGeometry` which has a value of `true`. Features that don't have an attribute named `lineGeometry` or that have a value other than `true` are left untouched.

```
FACTORY_DEF * TeeFactory \
    INPUT FEATURE_TYPE * lineGeometry true \
    OUTPUT FEATURE_TYPE * @GeometryType(fme_line)
```


@GeoreferenceRaster

```
@GeoreferenceRaster(POINT_AND_ANGLE, <X Upper Left Coordinate>,
<Y Upper Left Coordinate>, <X Spacing>, <Y Spacing>, <Rota-
tion>)

@GeoreferenceRaster(EXTENTS, <X Upper Left Coordinate>, <Y
Upper Left Coordinate>, <X Upper Right Coordinate>, <Y Upper
Right Coordinate>, <X Lower Right Coordinate>, <Y Lower Right
Coordinate>, <X Lower Left Coordinate>, <Y Lower Left Coordi-
nate>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
POINT_AND_ANGLE or EXTENTS	(POINT_AND_ANGLE EXTENT)	Specifies whether to accept spacing and rotation input or extent input. POINT_AND_ANGLE implies the upper left corner coordinates, the spacing, and the rotation are provided. EXTENT implies the upper left, upper right, lower right, and lower left corner coordinates are provided.	No
<X Upper Left Coordinate>	Reals	The upper left X coordinate of a raster	No
<Y Upper Left Coordinate>	Reals	The upper left Y coordinate of a raster	No
<X Spacing>	Reals	The horizontal length of a cell	No
<Y Spacing>	Reals	The vertical length of a cell	No
<Rotation>	Reals	The rotation of a raster, in degrees	No
<X Upper Right Coordinate>	Reals	The upper right X coordinate of a raster	No
<Y Upper Right Coordinate>	Reals	The upper right Y coordinate of a raster	No
<X Lower Right Coordinate>	Reals	The bottom right X coordinate of a raster	No
<Y Lower Right Coordinate>	Reals	The bottom right Y coordinate of a raster	No

Name	Range	Description	Optional
<Y Lower Left Coordinate>	Reals	The bottom left Y coordinate of a raster	No
<Y Lower Left Coordinate>	Reals	The bottom left Y coordinate of a raster	No

Configuration

This function does not accept configuration lines.

Description

The @GeoreferenceRaster function is used to georeference rasters. If the source raster is already georeferenced, this function will overwrite the previous georeference settings with the new ones.

When using POINT_AND_ANGLE, the raster's origin is set to the input upper left coordinate and its spacing and rotation are set to the input spacing and rotation; no calculations are done on the input data.

When using EXTENTS, all four input coordinates are validated to be unique, to form a rectangle which can be rotated, and to be clockwise. The rotation and spacing are then calculated from these coordinates. The origin of the raster is set to the input upper left coordinate.

Input features must contain raster geometries only.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a use of the @GeoreferenceRaster function. In this example, the SamplingFactory is used to pass through every feature and georeference each one. Note that for an input raster of size 10x10 (rows x columns), the provided examples will produce a raster with the same ground extent.

The first example georeferences input raster features to an upper left corner control point of (0,20) with both a vertical and horizontal spacing of 2 and a rotation of 45°.

```

FACTORY_DEF * SamplingFactory \
SAMPLE_RATE 1 \
INPUT FEATURE_TYPE * \
@GeoreferenceRaster(POINT_AND_ANGLE, 0, 20, 2, 2, 45) \

```


OUTPUT FEATURE_TYPE *

The next example georeferences input raster features to an upper left corner control point of (0,20), a upper right corner control point of (20,20), a lower right control point of (20,0), and a lower left control point of (0,0).

```
FACTORY_DEF * SamplingFactory \
    SAMPLE_RATE 1 \
    INPUT FEATURE_TYPE * \
        @GeoreferenceRaster(EXTENTS, 0, 20, 20, 20, 20, 0, 0, \
0) \
    OUTPUT FEATURE_TYPE *
```

@GlobalVariable

```
@GlobalVariable(<variable> [, <value>])
```

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<variable>	String	The name of the global variable.	No
<value>	Any Value	The value assigned to the variable.	Yes

Configuration

The @GlobalVariable function accepts two types of configuration lines:

`GlobalVariable <variable> <initialvalue>` – This configuration indicates that the initial value of the specified global variable be set to the given value. This will occur before the translation begins.

`GlobalVariable MAX_TO_LOG <number>` – This configuration configures the way the final state of all global variables is reported in the log messages. The global variables will be listed with their final values at the end of a translation.

If this configuration line is not present, a maximum of 50 global variables will be logged by default.

Name	Range	Description	Optional
<number>	Integer >= -1	Sets the maximum number of global variables that will be logged. If -1 is specified, all global variables are logged.	No

Description

The @GlobalVariable function is used to set and read global variables. Global variables are those that persist across all features during an FME translation run.

If this function is passed a single parameter, then the parameter is assumed to be a global variable name and the function will simply return the value of this variable. If the function is passed two parameters, then it will set the global variable with the first parameter's name to have the value of the second parameter.

A summary of the final values of all global variables used in the mapping file is output to the log file at the end of the translation.

If `fme_attribute_name` precedes either the variable or the value, the variable/value is instead treated as the name of an attribute that holds the information to use.

Inverse Operation

This function has no inverse.

Example

The following example uses the function both to set and to get values to offset all coordinates by a particular amount stored in a particular header record. The following sample assumes the header record comes before the features that have to be scaled.

```
# First set the global variables based on the value in the header
# records.
FEATURE_TYPE SamplingFactory \
  INPUT FEATURE_TYPE headerRecord \
    @GlobalVariable(XOffset, &xoffset) \
    @GlobalVariable(YOffset, &yoffset) \
  SAMPLE_RATE 0

# Now use the variables to scale all the other features
# that come by.
FEATURE_TYPE SamplingFactory \
  INPUT FEATURE_TYPE * \
    @Offset(@GlobalVariable(XOffset), @GlobalVariable(YOffset)) \
  SAMPLE_RATE 1
```

@GOID

```
@GOID()  
@GOID(VERIFY, <goid>)
```

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<goid>	32-Digit Hexadecimal Number	This parameter is used in the second form of @GOID to verify that a GOID is valid. The checksum of the <goid> is validated.	No (second form only)
<goid>	32-Digit Hexadecimal Number	This parameter is used in the second form of @GOID to verify that a GOID is valid. The checksum of the <goid> is validated.	No (second form only)

Configuration

The @GOID function does not accept configuration lines.

Description

This function calculates a Geographic Object Identifier (GOID) for the feature it is given, according to [1].

The GOID is a unique 128-bit number that incorporates the position of a feature with other numbers. The result is a unique value that may be attached to the feature, thereby distinguishing it from any other feature.

The first form of @GOID takes no parameters. In this case, it calculates a GOID for the feature, according to [1], and returns it as 32 hex digits.

There are two forms of the GOID function determined by the number of parameters given. They are:

Form 1: @GOID()

This form returns a 128-bit GOID for the feature as 32 hex digits in an ASCII string. The first 16 characters correspond to the position, the next 10 to the time, the next 4 to the sequence number, and the final 2 to the checksum. This form operates as an attribute function, because it returns a string.

For example, when @GOID was called on a point feature at position (-127,49) as measured in latitude and longitude on January 23, 1997 at 10:39:26 AM PST, it returned the value:

```
47B21A71B1BCC00013E280E5140000B1
```

Form 2: @GOID(VERIFY, <goidValue>)

This form verifies the `goidValue` expressed as a string of 32 hex digits is valid. The verification is done by ensuring the GOID's `checksum` is correct, according to the specification found in [1]. If the GOID checks out, 1 will be returned, otherwise 0 will be returned.

Inverse Operation

This function has no inverse and does nothing if invoked on the source line of a transfer specification.

Formal Specification

The GOID is composed of four parts:

- Part A corresponds to the left-most 64 bits,
- Part B corresponds to the next 40 bits,
- Part C corresponds to the following 16 bits, and
- Part D corresponds to the right-most 8 bits.

Part A

Sixty-four bits are derived from bit interleaving binary representations of the latitude and longitude values. They are initially expressed as double-precision floats. Next, the latitude and longitude are each multiplied by 107. Then each is converted to a signed, four-byte (32-bit) integer. Negative values are handled by two's complement encoding. The interleaving proceeds, from left to right, beginning with the left-most bit of the latitude and then the left-most bit of the longitude. Therefore, from left to right, the odd-numbered bits of Part A of the GOID are from the binary form of the latitude, and the even-numbered bits are from the binary form of the longitude.

The GOID provides approximately 1.1 cm precision at the equator for both coordinates. In British Columbia, Canada, the precision of the latitude and longitude are approximately 1.1 cm and better than 0.4 cm, respectively, for a given point defined as:

- (i) the position of the object, if its geometry is a point, or

- (ii) the weighted average of the first two points encountered in the object's geometric description.

In the case of an area feature, the two points come from the outer boundary as measured in a counterclockwise, or left-hand, fashion. The first point has double the weight of the second point to distinguish the traversal direction along the line segment.

That is, for both x and y coordinates:

```
new_coordinate = ((2*coordinate_first_point) +
coordinate_second_point) / 3
```

Part B

Forty bits represent time as measured in hundredths of a second, as read from the computer's clock. The time is measured with respect to Universal Coordinated Time (UTC), also known as Greenwich Mean Time (GMT), with a zero time of 00h00m00s, January 1, 1970, but unadjusted for leap seconds.

Part C

Sixteen bits are used as an arbitrary sequence number to avoid the possibility of two objects at the same place being defined in the same hundredth of a second. The sequence number is reset to zero with each FME run.

Part D

Eight bits are used as a checksum. For the purpose of this (Part D) calculation only, the highest four bits, bits 0 through 3, are moved to become bits 116 through 119. That is, the first, or left-most, hexadecimal digit becomes the 30th hexadecimal digit. The 128 bits are then treated as 16 consecutive, one-byte unsigned integers. The last integer is adjusted so the sum of the 16 integers is 0, modulo 256. This rotation of the first hex digit minimizes the chance that a four, 32-bit integer representation of the GOID could have the position of the integers altered or reversed in some way, thereby remaining undetected.

Encoding

There are two representations of the GOID: as a character string of 32 hexadecimal digits and as four unsigned, 32-bit integers.

- (i) The character string is defined from left to right, for Parts A through D, respectively. For each part, the left-most character is treated as the most significant. For Part A, this ensures that the correspondence between the hexadecimal and binary representations is direct.
- (i) Conversion from the ASCII representation to four 32-bit base-10 integers proceeds from left to right. The left-most 8 characters define the first integer, and so forth. Also, the left-most character in each group of 8 characters represents $n * 167$, the next represents $m * 166$, and so on for the remaining characters.

Example 1

In this example, each feature read from the input stream is assigned a GOID. The GOID is placed in the `goidString` attribute and can be further manipulated by the FME, or transferred to the output format to be stored with the feature.

```

FACTORY_DEF ARCGEN TeeFactory \
INPUT      FEATURE_TYPE * \
OUTPUT     FEATURE_TYPE * goidString @GOID()

```

Example 2

Some implementations may store and retrieve GOIDs as 32 bit unsigned integers. To use the GOIDs under these circumstances, a means of converting between the 32 bit unsigned integers and the 32 hex digits of the GOID is needed. The `@ConvertBase` function, combined with `@Split` and `@Concatenate`, provides the solution.

Case 1: Storing a newly computed GOID as four integers.

- 1 Compute the GOID and assign it to an attribute.
- 2 Split the GOID into four 8 hex digit pieces, assigning these to 4 attributes.
- 3 Use `@ConvertBase` to convert the four integers from 8 hex digit pieces to unsigned integers. Refer to `@ConvertBase` for details.

This can be done in two factory steps with:

```

FACTORY_DEF SAIF TeeFactory \
  INPUT FEATURE_TYPE * \
  OUTPUT FEATURE_TYPE * hexGoid @Goid() \
                        @Split(&hexGoid, "4s4s4s4s", hexPt1, \
                        hexPt2, hexPt3, hexPt4) \
FACTORY_DEF SAIF TeeFactory \
  INPUT FEATURE_TYPE * \
  OUTPUT FEATURE_TYPE * int1 @ConvertBase(&hexPt1, 16, 10) \
                        int2 @ConvertBase(&hexPt2, 16, 10) \
                        int3 @ConvertBase(&hexPt3, 16, 10) \
                        int4 @ConvertBase(&hexPt4, 16, 10)

```

Case 2: Retrieve a GOID stored as four integers, and covert into a hex string.

- 1 Convert each integer into hex. The example below assumes the integers were read from the input format in the attribute names <int1>, <int2>, <int3>, and <int4>.
- 2 Concatenate them together into the 32-bit string.
- 3 Validate the result, if you want, with the @GOID function.

This can all be done in one factory with:

```

FACTORY_DEF IGDS TestFactory \
  INPUT FEATURE_TYPE * hexPt1 @ConvertBase(&int1, 10, 16, 8) \
                        hexPt2 @ConvertBase(&int2, 10, 16, 8) \
                        hexPt3 @ConvertBase(&int3, 10, 16, 8) \
                        hexPt4 @ConvertBase(&int4, 10, 16, 8) \
                        goidHexString @Concatenate \
                        (&hexPt1, &hexPt2, &hexPt3, &hexPt4) \
  TEST @GOID(VERIFY, goidHexString) = 1 \
  OUTPUT PASSED FEATURE_TYPE * \
  OUTPUT FAILED FEATURE_TYPE * @Abort("Failed GOID validation \
-- &goidHexString is not a valid \
GOID")

```

References

- [1] GOID Specification, Subsection A.27.4, *Formal Specification*. Geographic Data BC.


```
@GridSnapper(<xMin>, <yMin>, <scale>, <output_int>, <zMin>,  
<zScale>) for 3D
```

Function Type: Attribute or Feature

Arguments:

Name	Range	Description	Optional
<xMin>	double	The false origin for x-values	No
<yMin>	double	The false origin for y-values	No
<scale>	double	A scale factor to convert to integers for x and y-values.	No
<output_int>	Yes/No	Select to view output as either the underlying integer ArcSDE® grid or the original plane coordinates.	No
<zMin>	double	The false origin for z-values	Yes
<zScale>	double	A scale factor to convert to integers for z-values.	Yes

Configuration

The @GridSnapper function does not accept configuration lines.

Description

The function will attempt to simulate the ArcSDE conversion on the feature, and will succeed only if all the coordinates of the feature are within the range defined by the ArcSDE format. However, illegal geometries, such as areas with dangles, will not be checked for validity.

All coordinates in the ArcSDE format must be 32-bit positive integers. Many coordinates do not meet this requirement; thus, an offset and scaling factor is required to convert all coordinates to integers. The conversion is performed with the equations

```
ArcSDE X = truncate ( ( ( X coordinate - <xMin> )
    * <scale> ) + 0.5 )
```

```
ArcSDE Y = truncate ( ( ( Y coordinate - <yMin> )
    * <scale> ) + 0.5 )
```

```
ArcSDE Z = truncate ((( Z coordinate - <zMin> )
* <zScale> ) + 0.5 )
```

If any adjacent coordinates snap onto the same location after conversion, the duplicates will be removed and will be recorded in a list attribute. If any of the coordinates are outside of the ArcSDE range, the conversion is cancelled and the function adds an error attribute to the feature.

If requested, ArcSDE coordinates are converted back to the original plane coordinates through the equations

```
X coordinate = (( ArcSDE X / <scale> ) + <xMin>)
Y coordinate = (( ArcSDE Y / <scale> ) + <yMin>)
Z coordinate = (( ArcSDE Z / <zScale> ) + <zMin>)
```

Arcs and ellipses passed through the GridSnapper will be stroked into lines; this matches the behaviour of the ArcSDE writer, which does not support the storage of these types of geometries.

Inverse Operation

This function has no inverse.

Example

In the example below, the input feature coordinates are converted to the ArcSDE format with a false origin of (5, 10, 15), a scale of 100, and a zScale of 1. The 'No' parameters indicates the feature will then be re-converted back to plane coordinates.

```
FACTORY_DEF * TeeFactory \
INPUT FEATURE_TYPE * \
OUTPUT FEATURE_TYPE * \
@OUTPUT FEATURE_TYPE * @GridSnapper(5,10,100,No,15,1)
```

@Http

```
@Http( GET|POST|DELETE|PUT, <url>, <target
attribute> [, <proxy info table name>] )
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<operation>	GET POST DELETE PUT	The download or upload operation requested. This must be one of GET, POST, DELETE, or PUT. Other HTTP operations are currently unsupported.	No
<url>	String	The target URL to perform the operation on.	No
<target attribute>	at- String	The results of the operation will be stored in this attribute.	No
<proxy info table name>	String	The name of the auxiliary information table. For instructions regarding construction of this table, please see the Configuration section.	Yes

Configuration

This function can accept an optional auxiliary table in order to store extra information which may be needed during translation.

```
HTTPRequest <proxy info table name>
[REQUEST_HEADER <request header>]*
[HTTP_USERNAME <http authentication username>]
[HTTP_PASSWORD <http authentication password>]
[HTTP_AUTHMETHOD <http authentication method>]
[PROXY_URL <proxy url>]
[PROXY_PORT <proxy port>]
[PROXY_USERNAME <proxy username>]
[PROXY_PASSWORD <proxy password>]
[PROXY_AUTHMETHOD <proxy authentication method>]
[UPLOAD_BODY <upload body>]
[UPLOAD_FILE <upload file>]
```

```
[UPLOAD_CONTENT_TYPE <upload content>]
[USE_RECV_HEADER_ENCODING <use header encoding>]
[CONTINUE_ON_ERROR <continue on error>]
```

Configuration Arguments:

Name	Range	Description	Optional
<proxy info table name>	String	The name of the auxiliary information table.	No
<request header>	String	A single custom request header enclosed in quotations. The REQUEST_HEADER tag may be used multiple times to generate multiple custom headers.	Yes
<http authentication username>	String	The HTTP basic access authentication username.	Yes
<http authentication password>	String	The HTTP basic access authentication password.	Yes
<http authentication method>	Basic Digest	The HTTP basic access authentication method.	Yes
<proxy url>	String	The target proxy URL.	Yes
<proxy port>	Integer	The target proxy port.	Yes
<proxy username>	String	The username for proxy authentication.	Yes
<proxy password>	String	The password for proxy authentication.	Yes
<proxy authentication method>	Basic Digest	The proxy authentication method to be used.	Yes
<upload body>	String	The message to be uploaded. To be used with POST and PUT only.	Yes
<upload file>	String	The path to the file that is to be uploaded. To be used with POST and PUT only.	Yes
<upload content>	String	The type of content that is to be uploaded. To be used only with POST and PUT.	Yes

Name	Range	Description	Optional
<code><use_header_coding></code>	en- YES NO	This specifies if the HTTP Response will be examined for encoding information or not. Defaults to NO.	Yes
<code><continue_on_error></code>	er- YES NO	This specifies if translation should continue in the event of an error. Defaults to NO.	Yes

Description

This function is responsible for performing the requested operation on the target URL, then reporting the results in an attribute. The HTTP Response status code will be stored in a separate `_http_status_code` attribute.

The first three parameters of the function are mandatory, providing the preferred operation, target URL, and chosen attribute to report the results. The fourth, optional parameter contains the name of an additional information table which handles more advanced settings such as proxy information.

Currently, only the HTTP GET, POST, DELETE, and PUT operations are available. Other HTTP operations are unsupported at this time. Specifying GET or DELETE will perform the associated operation on the URL and store the results in the target attribute. Specifying POST or PUT will upload the associated message or file as specified and store the results in the target attribute.

The HTTP basic access authentication is supported via the auxiliary configuration table for servers requiring credentials. The username, password and authentication method may be specified via the `HTTP_USERNAME`, `HTTP_PASSWORD`, and `HTTP_AUTHMETHOD`, respectively. Note that the HTTP basic access authentication is a mechanism designed to allow a client to provide credentials to a server on the assumption that the connection between them is trusted and secure. That is, any credentials passed from client to server can be easily intercepted through an insecure connection.

To specify a proxy, the auxiliary information table must be created. To avoid confusion, each table name should be unique. The appropriate proxy fields should be filled in order to use a proxy server, including `PROXY_URL`, `PROXY_PORT`, `PROXY_USERNAME`, `PROXY_PASSWORD`, and `PROXY_AUTHMETHOD` as necessary.

Custom request headers are supported. Custom request headers should follow the `REQUEST_HEADER` tag, enclosed in quotations. Each header should be separate, following its own `REQUEST_HEADER` tag.

To specify a file or message for upload, the appropriate fields must be completed. One of either a file or a message can be uploaded per function call. If a file is specified, it becomes the body of the HTTP request sent to the server. The content type provided will override the `Content-Type` header, regardless if one was provided as a custom header.

Should the `USE_RECV_HEADER_ENCODING` tag be set to YES, then the HTTP Response will be examined for encoding information so that the target attribute can be tagged with the encoding. If no encoding is found in the HTTP Response, the target attribute will be tagged as binary data. If this tag is set to NO, then the downloaded data will be considered text data in the current system encoding.

If an error occurs and the `CONTINUE_ON_ERROR` tag is set to YES, then the target attribute will be returned empty and an error message will be logged. Translation will still continue. If the tag is set to NO and an error occurs, the translation will fail and terminate.

Inverse Operation

This function does not have an inverse operation.

Example

```

HTTPRequest ExampleHttpRequest \
  REQUEST_HEADER "User-Agent: Safe Software FME" \
  REQUEST_HEADER "Accept-Encoding: gzip,deflate" \
  USE_RECV_HEADER_ENCODING YES \
  CONTINUE_ON_ERROR YES \

FACTORY_DEF * TeeFactory \
  FACTORY_NAME Example \
  INPUT FEATURE_TYPE * \
  OUTPUT FEATURE_TYPE * \
  @Http( GET, http://www.safe.com/, \
         _response, ExampleHttpRequest )

```

@Interpolate

@Interpolate(<startValue>,<endValue>)

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<startValue>	Number	The elevation value for the first vertex of the input feature.	No
<endValue>	Number	The elevation value for the last vertex of the input feature.	No

Configuration

This function does not accept configuration lines.

Description

This function interpolates elevation values along the length of a two-dimensional non-aggregated linear feature from a starting value to an ending value. Each vertex in the original input 2D line has an elevation value added to it. The elevation value is calculated so that the elevation is monotonically increasing or decreasing along the *run* of the line, in other words, the elevation is distributed between all the vertices so that the feature has a constant slope.

If the FME_GEOMETRY_HANDLING keyword is set to ‘no’ in the mapping file, an arc is demoted to its centerpoint prior to interpolation. If it is set to ‘yes’, the elevation of an arc is interpolated on its defined end-points and possibly mid-point (if the arc is an arc by 3 points).

Note: This function can also be used to distribute measure values across a feature for those destination systems which can retrieve measure data from the elevation (for example, the FME’s Shapefile writer can do this).

Inverse Operation

This function does nothing if invoked in the inverse direction.

Example

The following example interpolates elevation onto features which have a starting and ending elevation attribute:

```
FACTORY_DEF * TeeFactory \
```

```
INPUT FEATURE_TYPE *  
OUTPUT FEATURE_TYPE * @Interpolate(&startElev,&endElev)
```


@KeepAttributes

```
@KeepAttributes (<attrName>[ , <attrName>] *)  
  
@KeepAttributes (fme_regexp_match[ , <regexp>] +)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<attrName>	String	The name of the attribute to remain in the feature.	No
<regexp>	String	A regular expression that is used to match the names of the attributes being kept.	No

Configuration

The @KeepAttributes function does not accept configuration lines.

Description

This function clears all attributes from a feature, preserving only those whose names are provided as arguments. This function is used to reduce the number of attributes which are associated with a feature.

If the attribute name is a list attribute ending with {}, then all the attributes in this list are kept.

This function is only necessary if features are being created with an extremely large number of attributes (> 1000) and if the features' lifetimes are long due to being blocked in FME factories for processing. This function is also useful when transferring structures into SAIF where the SAIF definition of a structure is a subset of the features structure attribute. It may also be used in conjunction with AutoCAD's extended entity output to reduce the number of attributes stored.

If the first argument is fme_regexp_match, then all following parameters are interpreted as regular expressions (REs) and all attributes that match any of the REs will be kept.

RE Construction

The following rules determine one-character REs that match a single character:

- Any character that is not a special character (to be defined) matches itself.
- A backslash (\) followed by any special character matches the literal character itself (i.e., this "escapes" the special character).
- The "special characters" are:
+ * ? . [] ^ \$

The period (.) matches any character except the newline (for example, ".umpty" matches either "Humpty" or "Dumpty."

- A set of characters enclosed in brackets ([]) is a one-character RE that matches any of the characters in that set. For example, "[akm]" matches either an "a", "k", or "m". A range of characters can be indicated with a dash. For example, "[a-z]" matches any lowercase letter. However, if the first character of the set is the caret (^), then the RE matches any character *except* those in the set. It does not match the empty string. For example, "[^akm]" matches any character except "a", "k", or "m". The caret loses its special meaning if it is not the first character of the set.

The following rules can be used to build a multi-character RE:

- A one-character RE followed by an asterisk (*) matches zero or more occurrences of the RE. Hence, "[a-z]*" matches zero or more lowercase characters.
- A one-character RE followed by a plus (+) matches one or more occurrences of the RE. Hence, "[a-z]+" matches one or more lowercase characters.
- A question mark (?) is an optional element. The preceding RE can occur zero or once in the string – no more. For example, "xy?z" matches either "xyz" or "xz".
- The concatenation of REs is an RE that matches the corresponding concatenation of strings. For example, "[A-Z][a-z]*" matches any capitalized word.

Finally, the entire regular expression can be anchored to match only the beginning or end of a line:

- If the caret (^) is at the beginning of the RE, then the matched string must be at the beginning of a line.
- If the dollar sign (\$) is at the end of the RE, then the matched string must be at the end of the line.

Inverse Operation

This function does nothing when invoked in the reverse direction, which happens when it appears on the source portion of a transfer specification.

Example

When executed in the forward direction, the @KeepAttributes function is used to keep attributes `mif_type` and `mif_pen_width`. All features that enter the `ListFactory` are reduced to carrying only two attributes. The output is a single feature for every combination of `mif_type` and `mif_pen_width`.

```

FACTORY_DEF MIF ListFactory                                \
  INPUT FEATURE_TYPE bigFeature                            \
    @KeepAttributes(mif_type,mif_pen_width)                \
  GROUP_BY mif_type mif_pen_width                          \
  LIST_NAME dummyList{}                                    \
  OUTPUT POLYGON FEATURE_TYPE *
```

In the example below, any attributes that begin with `fme_` are kept on all `road` features. All other attributes are removed.

```

FACTORY_DEF SHAPE SamplingFactory                          \
  INPUT FEATURE_TYPE road                                  \
    @KeepAttributes(fme_regex_match, ^fme_)                \
  SAMPLE_RATE 1
```

@Length

```
@Length([<dimension> [, <multiplier>]])
```

```
@Length(TO_POINT, <dimension>, <point x>, <point y>
[, <point z>])
```

```
@Length(ALL_LENGTHS [, <dimension>
[, <multiplier>]])
```

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<dimension>	2 3	Specifies whether the 3 rd dimension is used in the length calculation. The default is 2, meaning only the x and y coordinates are used in the calculation. If 3 is specified and the feature has only 2 dimensions, no error will be flagged and the length will be calculated on the 2 available dimensions. If the first argument is TO_POINT then this argument is required.	Yes/No
<multiplier>	Real Number	By default, the length returned is in ground units. The multiplier, if specified, can be used to convert to other units. The default is 1.	Yes
<point x>	Real Number	The x coordinate for the vertex upto which the length is to be calculated. Used if the first argument is TO_POINT.	No
<point y>	Real Number	The y coordinate for the vertex upto which the length is to be calculated. Used if the first argument is TO_POINT.	No
<point z>	Real Number	The z coordinate for the vertex upto which the length is to be calculated. Used if the first argument is TO_POINT.	Yes

Configuration

This function does not accept configuration lines.

Description

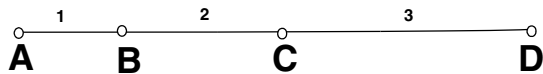
The @Length function calculates the length of features. For polygonal features, the length is equal to the sum of its perimeter and the perimeter of any holes within it. The optional multiplier is used to convert the returned value from ground units to units more useful to the caller.

If the first argument is set to TO_POINT then the function will calculate the length of feature upto the vertex specified by <point x>, <point y> [, <point z>]. Depending on the <dimension> argument, either a 2D or 3D length will be calculated.

If the first argument is set to ALL_LENGTHS then the function will return a comma-separated list of values, where each value is the distance from the start of the feature up to that vertex in the feature.

For example, a feature has 4 points: A, B, C and D.

AB, BC and CD are the distance between two consecutive vertices:



If $AB = 1$, $BC = 2$, and $CD = 3$ then the value returned by the function will be 0,1,3,6.

Inverse Operation

This function has no inverse.

Example

In the following example, the Shape len attribute is set to the 3D length of the SAIF Road when features are translated from SAIF to Shape. However, when features are translated from Shape to SAIF, the @Length call is ignored.

```
SHAPE roads    len @Length(3)
SAIF  Road::MOF
```

@Log

```
@Log ( [ <message> [ , <maxCoords> [ , maxFeatures> ] ] ] )
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<message>	Any String	An optional message may be specified. If present, this message is output to the log file before each feature is logged. This is useful to identify features if the @Log command is used in many places within a single mapping file.	Yes
<maxCoords>	Integer >= -1	The number of coordinates to be logged. If not specified, then the first 5 and the last 5 coordinates will be logged. If -1 is specified, then all coordinates will be logged. If 0 is specified, then no coordinates will be logged. If any other value is specified, then only the first <maxCoords> coordinates will be logged.	Yes
<maxFeatures>		The maximum number of features logged for this message. The number indicated by the LOG_MAX_FEATURES directive limits that given for maxFeatures .	Yes

Configuration

This function does not accept configuration lines.

Tip

Because the @Log function writes out large amounts of data to the log file, it significantly slows a translation and should be used only during testing and debugging. The LOG_MAX_FEATURES mapping file directive limits the number of features that may be logged in a single FME session.

Description

The @Log function outputs features to the FME log file. This function is primarily used during testing and debugging of transformation specifications, so the complete state of a feature can be viewed before and after it is transformed. It can also be used to log a feature before and after a feature function is applied.

This function is useful for logging erroneous features to the log file when the FME is used as a quality assurance tool for processing data.

Inverse Operation

The inverse is the same as the forward operation — in both cases, the feature is output to the log file.

Example

In the following example, the feature is logged as both a SAIF feature and a Design feature:

```
SAIF Contour::TRIM \
  position.geometry.value %value \
  @Generalize(10) @Log()

IGDS 12 igds_type igds_line igds_color 3 \
  @ZValue(%value) @Log()
```

In this second example, when Design is the destination, the Design feature is logged before and after the generalization function is applied. When a Design file is the source, the @Generalize function does nothing. The feature is logged twice and looks the same both times.

```
SAIF Contour::TRIM \
  position.geometry.value %value
IGDS 12 igds_type igds_line igds_color 3 \
  @ZValue(%value) \
  @Log("Before Generalization:") \
  @Generalize(10) \
  @Log("After Generalization:")
```

Note: Note the use of the optional message parameter to identify the logged features in the log file.

@Lookup

`@Lookup(<lut name>, <value> [, <flags>])`

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<lut name>	String	Identifies the lookup table. It must have been defined by a <code>Lookup</code> configuration line.	No
<value>	String	The value to be looked up in the named lookup table. The function will then return the found value, or will be blank if none was found.	No
<flags>	Constant value, Encoded attribute name	Configure how lookups are to be performed. The following values are supported: REVERSE ENCODED_ATTR REVERSE ENCODED_ATTR	Yes

Configuration

The `@Lookup` function accepts the following configuration line:

```
Lookup <lut name> [<source> <rep value>]+
```

Name	Range	Description	Optional
<lut name>	String	The name of the lookup table used as the first argument to the <code>@Lookup</code> function.	No
<source>	String	The value mapped to its replacement value when <code>@Lookup</code> is invoked in the forward direction.	No
<rep value>	String	The replacement value returned when the source value is passed to the <code>@Lookup</code> function invoked in the forward direction.	No

Description

The @Lookup function applies alternate values to source values through Lookup Tables (LUTs) during transformation. The LUTs are first defined in the mapping file on their own configuration lines. Many-to-one lookups are supported, but information is lost when the inverse operation is performed using such LUTs.

If a value to be looked up is not found in the table, then the translation will be aborted and the error will be reported. However, as such situations may occur legitimately, the special source value consisting of a blank string is the default. If a source value cannot be found in the list, then this default will be used if available. In addition, any occurrences of the special word `KEY` in the default replacement string are substituted with the original source value.

If the optional `REVERSE` parameter is specified, the direction of the lookup is reversed. That is, it will perform a lookup from replacement values to source values if operating in the forward direction, or from source value to replacement value if executing in the inverse direction.

If the optional `ENCODED_ATTR` is specified, the mapping direction is "FORWARD" and the `<value>` is expected to be an encoded attribute name. For more information on the encoding method, see *Substituting Strings in Mapping Files* in the *FME Fundamentals* help file (available at www.safe.com or via the Workbench and Universal Translator Help menus).

If the optional `REVERSE|ENCODED_ATTR` is specified, the mapping direction is "REVERSE" and the `<value>` is expected to be an encoded attribute name. For more information on the encoding method, see *Substituting Strings in Mapping Files* in the *FME Fundamentals* help file (available at www.safe.com or in the `<FME install directory>\help` folder).

The @Lookup function is an attribute value function, meaning that it is used to supply a value to an attribute, as opposed to modifying a whole feature.

Inverse Operation

When executed in the inverse direction on the source line of a transformation specification, the attribute's value is looked up among the LUT's replacement values and the transfer variable is assigned the appropriate source value. This allows the same lookup statement to be used for both directions of a translation.

If the LUT mapped several source values to the same replacement value, information will be lost on the inverse operation. Only the last source value corresponding to the replacement value is output.

Example 1

In the following example, the `parkTypeLut` defines a LUT from codes used as Shape file attributes to their SAIF enumeration equivalents. When SAIF is the destination system, the value of the `%pk` transfer variable is looked up in the table and its mapping is returned. For example, if a Shape feature had a `pkType` of `HP`, `@Lookup` would return `historicPark`.

When translation occurs from SAIF back to Shape, `@Lookup` runs in the reverse direction. In this case, it scans the right-hand side of the LUT for a match to the value of the SAIF feature's `parkType` attribute. When a match is found, it sets the `%pk` transfer variable to the left-hand side LUT value. For example, if a SAIF feature had a `parkType` value of `nationalPark`, `%pk` would be set to `NP`.

```
# -----
# Define the parkType lookup table. It maps values
# held in a shape file to their SAIF enumeration tag
Lookup parkTypeLut  HP      historicPark          \
                    NE      naturalEnvironmentPark \
                    NP      nationalPark           \
                    PA      protectedArea
SHAPE parks      pkType %pk
SAIF  Park::MOF position.geometry.Class Point      \
                    parkType @Lookup(parkTypeLut,%pk)
```

Example 2

In this example, the default lookup is used, allowing data outside of the source value's expected range to pass through without causing the translation to abort. The output data can then be analyzed to see which source values were not handled by the LUT by selecting all records that have the string "Unknown Ownership Code" in them.

```
Lookup ownershipCodeLut \
  NM  "Northwest Mounted Police" \
  FW  "Fisheries and Wildlife" \
  MF  "Forestry" \
  " " "Unknown Ownership Code KEY"
SHAPE parcels  OWNCODE %code
MIF  mifparcel  ownership @Lookup(ownershipCodeLut,%code)
```

@MergeLists

```
@MergeLists(<destListName>,<sourceList1>[,<sourceList2> ...])
```

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<destListName>	Any Feature Attribute List	The function will create a new attribute list of this name. The name should contain a {} pair to indicate that it is a list.	No
<sourceListN>	Any Feature Attribute List	Any number of list names may be indicated. All the lists given will be merged together in the order they are listed. All elements in nested lists are also merged. The names should contain a {} pair to indicate that they are lists.	Must have at least one source list

Configuration

The @MergeLists function does not accept configuration lines.

Description

This function creates a new list attribute that has all the elements of several other lists merged together in the order they are given. All elements from both simple and nested lists are merged in order.

Inverse Operation

This function has no inverse.

Example

Suppose one has a feature that has several attribute lists that give ID numbers of lines that form a donut polygon. One simple list has all the ID numbers of the lines that form the outer edge of the polygon, while a nested list holds the ID numbers of all the lines that form holes within the polygon. (Such a feature may be returned from the AutoKa PC FF reader.) We can merge all the ID numbers

FME Functions and Factories



Feature Type: polygon	
Attribute Name	Value
ff_all_edges{0}	7
ff_all_edges{1}	12
ff_all_edges{2}	4
ff_all_edges{3}	16
ff_all_edges{4}	8
ff_all_edges{5}	3

Feature Type: polygon	
Attribute Name	Value
ff_all_edges{6}	9
ff_all_edges{7}	11
ff_all_edges{8}	20
ff_all_edges{9}	1
ff_all_edges{10}	5
ff_all_edges{11}	14
ff_all_edges{12}	10
ff_all_edges{13}	2
ff_all_edges{14}	13
ff_all_edges{15}	6

@MGRS

```
@MGRS(TO_MGRS, <Ellipsoid name>, <lettering type>,  
<precision>, [x, y])  
  
@MGRS(FROM_MGRS, <Ellipsoid name>, <lettering type>,  
<MGRS code>, [x, y])
```

Function Type: Attribute or Feature

Arguments:

Name	Range	Description	Optional
<Ellipsoid name>	Any Ellipsoid name supported by FME	The Ellipsoid used for the conversion to or from MGRS.	No
<lettering type>	WGS84_LETTERING BESSEL_LETTERING	The type of lettering used	No
<precision>	Integer from 0 to 5 (inclusive)	The number of digits used for eastings and northings in the MGRS code. A precision of 5 locates a point within 1 meter.	No
<MGRS code>	MGRS String	An MGRS code used to convert to lat/long coordinates.	No
[x, y]	x : [-180, 180] y : (-90, 90) Any String	In the TO_MGRS case, these parameters are the lat/long coordinates used for the conversion to MGRS code. In the FROM_MGRS case, these parameters are used as the names of the attributes that will store the lat/long coordinates once the MGRS code is converted. Note: These parameters must BOTH be used or BOTH not used. Passing in only one of the parameters will result in an error.	Yes

Configuration

The @MGRS function does not accept configuration lines.

Description

This function converts between latitude / longitude coordinates and Military Grid Reference System (MGRS) code. It behaves differently depending on whether the optional parameters are included.

The Military Grid Reference System (MGRS) is an extension of the UTM system. The MGRS code for a position consists of a group of letters and numbers which include the following elements:

- The Grid Zone Designation.
- The 100,000-meter square letter identification.
- The grid coordinates (also referred to as rectangular coordinates); the numerical portion of the reference expressed to a desired refinement.
- A reference is written as an entity without spaces, parentheses, dashes, or decimal points.

Examples:

- 18S (Locating a point within the Grid Zone Designation)
- 18SUU (Locating a point within a 100,000-meter square) - Precision of 0
- 18SUU80 (Locating a point within a 10,000-meter square) - Precision of 1
- 18SUU8401 (Locating a point within a 1,000-meter square) - Precision of 2
- 18SUU836014 (Locating a point within a 100-meter square) - Precision of 3
- 18SUU83630143 (Locating a point within a 10-meter square) - Precision of 4
- 18SUU8362601432 (Locating a point within a 1-meter square) - Precision of 5

In the TO_MGRS conversion, there are two major cases:

- 1 If the x and y (optional) parameters are NOT passed in, then the coordinates that will be used in the conversion to an MGRS code are those of the feature.
NOTE: If a non-point feature (i.e. a feature with more than one coordinate) is passed in, then the first point in the data will be used for the conversion.
- 2 If the x and y (optional) parameters ARE passed in, then the values of these parameters are the lat/long values that will be used to calculate the MGRS code.

In both of the above cases, the MGRS code is the return value of the function.

In the `FROM_MGRS` conversion, there are, again, two major cases:

- 1 If the `x` and `y` (optional) parameters are NOT passed in, then the feature will be set to be a point at the lat/long values obtained from the MGRS code. The point's coordinate system is set to be the one that was passed in to the function.
- 2 If the `x` and `y` (optional) parameters ARE passed in, then the values of these parameters will be used as the names of the attributes that will hold the converted lat/long values. The only change to the feature is the addition of these attributes.

In both of the above cases, the function returns no value.

Note Conversion to and from MGRS may cause some shift, but only the first time. Rounding of the results to some known amount of precision may be desired (only when converting from MGRS back to lat/long).

Inverse Operation

This function has no inverse.

Example

In the example below, the input feature geometry's coordinates are converted from lat/long to an MGRS code of precision to 1 meter, using WGS84 Ellipsoid and WGS84 lettering. The MGRS code is stored in an attribute called `_MGRS`.

```
FACTORY_DEF * TeeFactory \
INPUT FEATURE_TYPE * \
OUTPUT FEATURE_TYPE * _MGRS @MGRS(TO_MGRS, LL-83, WGS84_LETTERING, 5)
```


@MinimumSpanningCircle

```
@MinimumSpanningCircle(REPLACE_GEOM)

or

@MinimumSpanningCircle(SET_ATTR, <xCenterAttrName>,
<yCenterAttrName>, <radiusAttrName>)
```

Function Type: Attribute or Feature

Arguments:

Name	Range		Description	Optional
<xCenterAttr-Name>	any name	attribute	The name of the attribute into which the x-value of the circle center will be placed.	No
<yCenterAttr-Name>	any name	attribute	The name of the attribute into which the y-value of the circle center will be placed.	No
<radiusAttr-Name>	any name	attribute	The name of the attribute into which the radius of the circle center will be placed.	No

Configuration

The @MinimumSpanningCircle function does not accept configuration lines.

Description

If REPLACE_GEOM is specified, the function converts the geometry of the feature to the minimum spanning circle.

If SET_ATTR is specified, the function simply adds three new attributes to the feature using the attributes specified by the user.

Inverse Operation

This function has no inverse.

Example

In the example below, the convex hull geometry of the feature is converted to the minimum spanning circle.

```
FACTORY_DEF * TeeFactory \
  INPUT FEATURE_TYPE * \
  OUTPUT FEATURE_TYPE * \
    @MinimumSpanningCircle(REPLACE_GEOM)
```

@NumCoords

```
@NumCoords ( [ FLATTEN_AGGREGATE ] )
```

Function Type: Attribute

Arguments

One optional argument `FLATTEN_AGGREGATE` for multi-part or aggregate features. This flag will ensure that the the actual number of coordinates are returned for the aggregate feature as suppose to the actual storage used for the feature. An aggregate feature may use more more storage space than the actual coordinates of the feature. The number of coordinates returned with `FLATTEN_AGGREGATE` will exclude any internal FME meta information.

Configuration

The `@NumCoords` function does not accept configuration lines.

Description

This function returns the number of coordinates that define the feature's geometry.

Inverse Operation

The function performs the same operation in the inverse direction.

Example

In the example below, the MapInfo `numCoords` attribute is set to the number of coordinates stored in the feature when going from Shape to MIF. When going from MIF to Shape, the command has no effect.

```
SHAPE lake
MIF lake numCoords @NumCoords()
```

@NumElements

@NumElements (<listName>)

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<listName>	Any Feature Attribute List	The function returns the number of elements present in this attribute list. The name should contain a { } pair to indicate that it is a list.	No

Configuration

The @NumElements function does not accept configuration lines.

Description

This function returns a count of the number of elements in an attribute list. Certain legacy systems, most often those using the Column Aligned Text (CAT) format, require such a count to be output. Consequently, this function is most commonly used to compute a value for a TRANSFER clause in a relation definition. Refer to the @Relate section in this chapter for details.

Inverse Operation

This function has no inverse.

Example

Given the following feature:



Feature Type: ForestStand::BCFOR	
Attribute Name	Value
stand{0}.species	birch
stand{0}.age	30
stand{1}.species	birch
stand{1}.age	30
Coordinates: 50321, 5022321, 35	

`@NumElements(stand{ })` would return 2, since there are two elements (`stand{0}` and `stand{1}`) in the `stand` attribute list.

The 1:M example in the `@Relate` section of this chapter contains a complete example of `@NumElements` used in conjunction with the `@Relate` function.

@NumHoles

@NumHoles()

Function Type: Attribute

Arguments: None

Configuration

The @NumHoles function does not accept configuration lines.

Description

This function returns the number of holes in a polygonal feature. If the feature was not polygonal, or it had no holes, it returns 0. If the feature contained disjoint polygons, then the number returned is the total number of holes in all pieces of the feature.

Inverse

When invoked on the source line of a transfer specification, @NumHoles does nothing.

Example

In this example, all polygons read from a Shape file are tested to see if they have holes. Those with holes are given a feature type on output of `hasHoles`, while those without are given the feature type `noHoles`.

```

FACTORY_DEF SHAPE TestFactory                                \
    INPUT FEATURE_TYPE * SHAPE_GEOMETRY shape_polygon      \
    TEST @NumHoles() > 0                                     \
    OUTPUT PASSED FEATURE_TYPE hasHoles                      \
    OUTPUT FAILED FEATURE_TYPE noHoles

```

@Offset

```
@Offset(<offset>)
@Offset(<xOffset>, <yOffset>)
@Offset(<xOffset>, <yOffset>, <zOffset>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<offset>	Real Value	All X, Y, and Z coordinates are offset by this value.	No
<xOffset>	Real Value	The X offset applied to the feature.	No
<yOffset>	Real Value	The Y offset applied to the feature.	No
<zOffset>	Real Value	The Z offset applied to the feature.	No

Configuration

The @Offset function does not accept configuration lines.

Description

This command adds the offsets to the feature's coordinates. If just one value is specified, then X, Y, and Z are all offset by that amount. If two values are specified, then X and Y are offset as specified and the Z component is left untouched. If three values are specified, then X, Y, and Z are offset as specified.

This command also supports raster features and supports sub-selection of raster bands and palettes.

Inverse Operation

The function subtracts the offsets from the feature coordinates.

Example

In the example below, the building coordinates are offset by 100 when going from Shape to MIF. When going from MIF to Shape, the building coordinates are subtracted by 100.

```
SHAPE building
MIF building @Offset(100.0)
```


@OGCGeometry

Note: For a method of storing and retrieving geometry in an attribute where the geometry is fully preserved (including measures and complex chains involving arcs), see @Geometry.

```
@OGCGeometry(<command>, <ogcType>, <attrName>, <ogc-version>)
```

```
@OGCGeometry(<validationType>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<command>	(from_attribute to_attribute)	To specify whether to import or export the geometry of a feature from/to an attribute.	No
<ogcType>	(wkb wkt wkbhex)	The string specifying the OGC format of the geometry.	No
<attrName>	String	The name of the attribute to have the geometry saved to or retrieved from the feature.	No
<ogcversion>	String (1.1 1.2)	When exporting (to_attribute), specifies which version of OGC to use. Currently supported are versions 1.1 and 1.2; 1.2 includes measure support. The default is 1.1.	No
<validation-Type>	(is_simple is_valid)	The string specifying the type of validation query to be performed on the input geometry.	No

Configuration

This function does not accept configuration lines.

Description

This command is used to set, get or validate the feature geometry. In both get and set cases, the OGC type can be set to either Well-Known Text (wkt), Well-Known Binary (wkb), or Well-Known Binary encoded as hexadecimal digits (wkbhex). Depending on the <command> string, the attribute given will be the

source or destination for the geometry of the feature. For the get and set modes the validation type is irrelevant.

OGC version 1.2 adds support for measures on geometries. In `from_attribute` mode this means any measures given in the input (such as in `POLYGON M`) will be put onto the generated geometry as its “default” (unnamed) measure. In `to_attribute` mode if the input geometry has a default measure it will be added to the output WKT or WKB.

Note that in `to_attribute` mode, if the feature has no geometry, then the resulting attribute will have a blank value, because the OGC specifications provide no way to represent a “null” geometry. Conversely, however, in `from_attribute` mode, if the specified attribute has a blank value, then the feature’s geometry will be left untouched and a warning output.

When using the validation type, all other options need not be specified. This function can either validate geometry as being simple or not, or in a more basic sense, whether it is a valid OGC geometry or not according to the two options. If the geometry is invalid, attributes will be added to the feature which describe where and why it is invalid. These attributes are as follows:

Attribute	Description
<code>_validateFailCoordX</code>	X coordinate where the geometry is invalid
<code>_validateFailCoordY</code>	Y coordinate where the geometry is invalid
<code>_validateFailReason</code>	Failure case index (an integer)
<code>_validateFailReasonString</code>	English description of the failure

The possible values of `_validateFailReason` and their corresponding descriptions are:

<code>_validateFailReason</code>	<code>_validateFailReasonString</code>
0	Undetermined Error
1	Repeated Point*
2	Hole Outside Shell
3	Nested Holes
4	Disconnected Interior
5	Self Intersection
6	Ring Self Intersection
7	Nested Shells
8	Duplicated Rings
9	Too Few Points
10	Invalid Coordinate
11	Ring Not Closed

* Not currently implemented.

Inverse Operation

The command has no effect in the inverse direction.

Example

The following example shows a use of the @OGCGeometry function. In this example, the `SamplingFactory` is used to filter the road features, the geometry are converted to Well-Known Text, and stored into the attribute `featGeom`.

```

FACTORY_DEF * SamplingFactory                                \
    SAMPLE_RATE 1                                           \
    INPUT FEATURE_TYPE road                                \
    @OGCGeometry(to_attribute, wkt, featGeom, 1.1)         \
    OUTPUT FEATURE_TYPE *
```

The following example shows a use of the @OGCGeometry function to determine whether or not the input features are valid. In this example, the `TestFactory` is used to filter the road features into either valid or invalid features.

```

FACTORY_DEF * TestFactory
    FACTORY_NAME VALID_GEOM_TESTER
    INPUT FEATURE_TYPE road
    TEST @OGCGeometry(is_valid) == true
    OUTPUT PASSED FEATURE_TYPE VALID_GEOM
    OUTPUT FAILED FEATURE_TYPE INVALID_GEOM
```


@OrderRasterBands

```
@OrderRasterBands (<band list>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<band list>	String	A space-delimited string that specifies the desired order of the bands.	No

Configuration

This function does not accept configuration lines.

Description

The `@OrderRasterBands` function is used to change the order of bands in a raster.

The `<band list>` parameter is a list of band indices, separated by spaces, which specifies the desired order of the bands. Indices are zero-based, so the first band is at index 0. Bands in the original raster that are not specified in the string will be appended after all the specified bands, in their original order. Note that bands may not be included in the list more than once.

For example, given an RGBA input raster with four bands and a band list of "3 0 2", the resulting raster would have bands ARBG.

Input features must contain raster geometries only.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a typical use of the `@OrderRasterBands` function. In this example, the `SamplingFactory` is used to pass through every feature, while the `@OrderRasterBands` function reorders the bands such that the first two bands of the raster swap positions, while the third band remains in that same position (and any other bands will be appended afterwards).

```
FACTORY_DEF * SamplingFactory \
SAMPLE_RATE 1 \
```

```
INPUT FEATURE_TYPE *  
    @OrderRasterBands(1 0 2)  
OUTPUT FEATURE_TYPE *
```

```
\  
\
```

@Orient

```
@Orient(<orientation>)
@Orient()
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<orientation>	RIGHT_HAND_RULE LEFT_HAND_RULE REVERSE	This parameter controls what orientation the feature has when the function is complete.	No

Configuration

This function does not accept configuration lines.

Description

This function adjusts the orientation of a polygonal feature or the direction of a linear feature.

@Orient takes a single parameter that specifies the orientation the feature will have when the function completes. For the outer boundary, this means that the vertices are in a clockwise direction whereas for holes, the vertices are in a counterclockwise direction.

The opposite is true when LEFT_HAND_RULE is specified. When the LEFT_HAND_RULE is specified, the feature is returned so that the outer boundary's vertices are in a counterclockwise order and the holes have a clockwise ordering.

When REVERSE is specified, the feature's coordinates are flipped so that the first coordinate becomes the last one, and vice-versa. This option is intended for use with features that are not polygonal.

If no parameter is given, the function does not modify the feature, but instead returns the feature's existing orientation: right_hand_rule, left_hand_rule, or no_orientation. This last value will result if the feature is non-polygonal, is a donut whose shells' orientations are mixed, or is an aggregate with either no polygonal parts or polygonal parts of mixed orientations.

Inverse

When invoked on the source line of a transfer specification, `@Orient` orients the feature in the opposite direction to the parameter it is passed.

Example:

In this example, all polygons read from a MIF file are adjusted to follow right-hand-rule orientation.

```
FACTORY_DEF      SHAPE SamplingFactory          \  
                  SAMPLE_RATE 1                  \  
                  INPUT FEATURE_TYPE *mif_type mif_region  \  
                                @Orient(RIGHT_HAND_RULE)
```


@ProcessRasterTiles

```
@ProcessRasterTiles (<mode>,<pattern>[,<num tile
rows>,<num tile cols>])
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<mode>	DESTROY MINMAX	A string that specifies what mode to set the function in. DESTROY reads all the tile(s) of the raster. MINMAX adds attributes relating to the minimum and maximum properties of the bands and palettes.	No
<pattern>	BAND TILE	A string that specifies what pattern to read the tiles in. BAND means read each band at a time (BSQ). Tiles may be used to partition each band into manageable chunks. TILE means read a of the raster tile at a time until all tiles are consumed. Several bands may be read for a given tile.	No
<num tile rows>	Unsigned integer	An unsigned integer that specifies the number of tile rows. A value of 0 defaults to the size reported by the band. This is only used in DESTROY mode.	Yes
<num tile cols>	Unsigned integer	An unsigned integer that specifies the number of tile columns. A value of 0 defaults to the size reported by the band. This is only used in DESTROY mode.	Yes

Configuration

This function does not accept configuration lines.

Description

The `@ProcessRasterTiles` function processes all the tile(s) of the raster. The `<mode>` parameter specifies what processing is done on the input.

In `DESTROY` mode, all the tile(s) of the raster are requested, but no actual operations are performed on the tile(s). Note that this mode is mostly for testing performance and code path execution or for narrowing down problems in a workspace by eliminating the writer. This mode may not be very useful in most translation scenarios. Two optional parameters can be used to specify the tile's number of rows and columns, both of which must be set for them to be taken into account.

In `MINMAX` mode, the minimum and maximum of each band and palette are added as attributes to the raster. The following band and palette attributes become exposed:

- `_band{}.min`
- `_band{}.max`
- `_band{}.palette{}.keyMin`
- `_band{}.palette{}.keyMax`
- `_band{}.palette{}.valueMin`
- `_band{}.palette{}.valueMax`

Note that wherever “{}” occurs above, there will be one instance of the attribute for each band or palette. The first will appear with {0}, the second with {1}, and so on.

Any nodata values that are present on the bands of the raster are ignored in the minimum and maximum value calculations.

Input features must contain raster geometries only.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example 1

The following example shows a typical use of the `@ProcessRasterTiles` function in `DESTROY` mode. In this example, the `SamplingFactory` is used to pass through every feature while the `@ProcessRasterTiles` requests all the tile(s) of the raster without operating on the tile(s).

```
FACTORY_DEF * SamplingFactory \
SAMPLE_RATE 1 \
```

```
INPUT FEATURE_TYPE *
    @ProcessRasterTiles (DESTROY)
OUTPUT FEATURE_TYPE *
```

@Python

Note: This function is not supported by FME Base Edition.

```
@Python(<moduleName>.<functionName> [, <argument>]+)
```

Function Type: Attribute or Feature

Arguments:

Name	Range	Description	Optional
<moduleName>	String	The name of the module where the function resides.	No
<functionName>	String	The name of the Python function to call.	No
<argument>+	String	One or more arguments to pass to the Python function. Attribute values may be used as arguments by prefixing the attribute name with an ampersand ('&').	Yes

Configuration

This function does not accept configuration lines.

Description

This FME function executes an arbitrary Python 2.3 or 2.4 function, and returns the result. It allows access to a complete procedural language with rich built-in primitives from within FME mapping files. Python procedures can be written to simplify tasks that are awkward to accomplish using standard mapping file syntax, and to allow users to extend FME to solve new problems.

To use this function effectively, familiarity with the Python language is required. There are several sources of information available to assist in learning Python. The Internet has many Python sites – the official site is at <http://www.python.org>.

Note FME does not embed a Python interpreter, and requires a Python interpreter to be installed on the system in order for this function to work. Python distributions are available from <http://python.org> and <http://activestate.com>

Python Function

The Python function that Python calls must, at the very minimum, accept a feature object as its first parameter. The feature is provided as an `FMEFeature` python object. The `FMEFeature` Python class is included as part of `pyfme`, FME's Python FME Objects API. For further documentation on `FMEFeature`'s available methods, please see `$(FME_HOME)/fmeobjects/python/apidoc`.

Python functions called by `@Python` can, if desired, accept any number of optional arguments.

Examples:

```
# This function returns the contents of the test attribute.
# The FME call looks like:
#   @Python(examples.noArgs)
def noArgs(feature):
    return feature.getStringAttribute('test')

# This function sets two attributes, and returns the sum
# of two arguments. The FME call looks like:
#   @Python(examples.addAB,1,2)
def addAB(feature,a,b):
    feature.setIntAttribute(int(a))
    feature.setIntAttribute(int(b))
    c = int(a) + int(b)
    return str(c)
```

FME Objects Logging

It is possible to directly access the FME log file from a Python function via an `FMELogfile` object.

Example:

```
def logExample(feature,mymessage):
    import pyfme
    logger = pyfme.FMELogfile()
    logger.log(mymessage,pyfme.FME_INFORM)

@Python( examples.logExample, "A message to log" )
```

Interpreter Configuration

The following directives can be used to configure the Python interpreter that FME loads.

Name	Value
FME_PYTHON_PATH	A path to add to Python's <code>sys.path</code> array that controls module loading. Wrap the path with quotation marks if it contains white space.
FME_PYTHON_IMPORT	The name of a Python module to import. Note: The module must reside in a directory referenced by python's <code>sys.path</code> array.

Inverse Operation

This function has no inverse.

Example

In the example below, a Python function is used to calculate the boundaries of a section of land given an FME feature that has the boundaries of its containing township. The Python function, `bounds`, is defined in the file `sections.py` that resides in the same directory as the calling mapping file.

```
# -----
# sections.py
# Borrowed from the @Tcl2 Township section example
#
# Compute the bounds of the section, which is passed in to
# the function as its only argument.
# This assumes that the feature already has attribute entries in it for the
#   containing township bounds (townshipMinX,
# townshipMinY, townshipHeight, townshipWidth)

# The section grid looks like this:
# 31 32 33 34 35 36 ROW 5
# 30 29 28 27 26 25 ROW 4
# 19 20 21 22 23 24 ROW 3
# 18 17 16 15 14 13 ROW 2
#  7  8  9 10 11 12 ROW 1
#  6  5  4  3  2  1 ROW 0
#
#  0  1  2  3  4  5 COLUMN

import pyfme

def bounds(feature,section):

    # First determine the size of a section in the township
    sectionWidth  = float(feature.townshipWidth)/6.0
    sectionHeight = float(feature.townshipHeight)/6.0
```

```

# Now figure out which row and column of the section in the township.
row = (int(section)-1) / 6
col = (int(section)-1) % 6

# Adjust for even rows, when the numbering goes the wrong way
if (row % 2) == 0: col = 5 - col

# Now calculate the min and max corners for the section
minX = col * sectionWidth + float(feature.townshipMinX)
maxX = minX + sectionWidth
minY = col * sectionHeight + float(feature.townshipMinY)
maxY = minY + sectionHeight

# And set them as attributes in the FME feature
feature.setRealAttribute('sectionMinX',minX)
feature.setRealAttribute('sectionMaxX',maxX)
feature.setRealAttribute('sectionMinY',minY)
feature.setRealAttribute('sectionMaxY',maxY)
feature.setRealAttribute('sectionHeight',sectionHeight)
feature.setRealAttribute('sectionWidth',sectionWidth)

# And finally set the coordinates of the feature to these bounds
feature.resetCoords()
feature.setDimension( 2 )
feature.addCoordinates( (minX,minX,maxX,maxX,minX),
(minY,maxY,maxY,minY,minY) )
feature.setGeometryType( pyfme.FME_GEOM_POLYGON )

return None

# end sections.py
# -----

# -----
# sections.fme
FME_PYTHON_IMPORT sections
FME_PYTHON_PATH "$(FME_MF_DIR) "

READER_TYPE NULL
WRITER_TYPE NULL
NULL_DATASET null

FACTORY_DEF * CreationFactory
    OUTPUT FEATURE_TYPE sectionFeature \
        @SupplyAttributes(townshipWidth,6) \
        @SupplyAttributes(townshipHeight,12) \
        @SupplyAttributes(townshipMinX,400) \
        @SupplyAttributes(townshipMinY,100) \
        @SupplyAttributes(section,6) \
        @Log("Before") \
        @Python(sections.bounds,6) \
        @Log("After")

```

After the `@Python` function is called, the feature will have the various section attributes present on it (`sectionMaxY`, `sectionMinY`, `sectionMaxX`, `sectionMinX`, `sectionWidth`, `sectionHeight`). Its geometry will also be set to the section bounds polygon.

@RasterCellOrigin

@RasterCellOrigin(<X cell origin>, <Y cell origin>)

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<X cell origin>	[0-1]	The X component of the cell or- igin.	No
<Y cell origin>	[0-1]	The Y component of the cell or- igin.	No

Configuration

This function does not accept configuration lines.

Description

The @RasterCellOrigin function is used to set a raster’s cell origin. This can be used to specify whether each cell’s data point is at the lower-left or center (or somewhere else) within the cell.

The <X cell origin> and <Y cell origin> parameters are used to set the raster’s cell origin. Each of these parameters must be between 0.0 and 1.0 inclusively.

Input features must contain raster geometries only.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a typical use of the @RasterCellOrigin function. In this example, the SamplingFactory is used to pass through every feature, while the @RasterCellOrigin function sets the rasters’ cell origin to (0.5, 0.5).

```
FACTORY_DEF * SamplingFactory \
    SAMPLE_RATE 1 \
    INPUT FEATURE_TYPE * \
    @RasterCellOrigin(0.5, 0.5) \
    OUTPUT FEATURE_TYPE *
```


@RasterGCP

```
@RasterGCP(<mode>[,<coord_sys>,<gcp_value>])
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<mode>	SET GET	A string that specifies whether the function will set the GCP on the raster or retrieve the GCP from the raster	No
<coord_sys>	Source Coord. System	A string value specifying the source coordinate system. Optional if the <mode> is GET.	Yes
<gcp_value>	UInt UInt Reals Reals Reals	A string value specifying the GCP(s). Each GCP is separated with a semi-colon and each value of a 5-value GCP is separated by a space. Optional if the <mode> is GET.	Yes

Configuration

This function does not accept configuration lines.

Description

The @RasterGCP function is used to set or get the Ground Control Points (GCP) from the raster. The <gcp_value> is formatted as a 5 value string. Each GCP is separated with a semi-colon and each value in a GCP is separated by a space. Below is an example of a <gcp_value>:

```
pixel(column) line(row) X-Coord Y-Coord Z-Coord
```

In SET mode, <coord_sys> and <gcp_value> must be specified. This mode sets GCP(s) onto the raster. If the raster already has existing GCP(s), they will be overwritten with the new GCP(s).

In GET mode, both the <coord_sys> and <gcp_value> are optional parameters. This function retrieves the GCP(s) from the raster.

Input features must contain raster geometries only.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example 1

The following example shows a typical use of the @RasterGCP function in “SET” mode. In this example, the `SamplingFactory` is used to pass through every feature while the @RasterGCP function sets a GCP value on pixel 3, line 5 with (35.2,23.4,-30). The source coordinate system is UTM84-17N

```

FACTORY_DEF * SamplingFactory           \
SAMPLE_RATE 1                           \
INPUT FEATURE_TYPE *                     \
    @RasterGCP(SET, UTM84-17N, 3 5 35.2 23.4 -30) \
OUTPUT FEATURE_TYPE *

```

Example 2

The following example shows a typical use of the @RasterGCP function in “GET” mode. In this example, the `SamplingFactory` is used to pass through every feature while the @RasterGCP function gets the GCP value on the raster.

```

FACTORY_DEF * SamplingFactory           \
SAMPLE_RATE 1                           \
INPUT FEATURE_TYPE *                     \
    @RasterGCP(GET)                       \
OUTPUT FEATURE_TYPE *

```

@RasterGeometry

```
@RasterGeometry(EXTRACT, <format>, <formatAttribute>,
<blobAttribute>)

@RasterGeometry(REPLACE, <format>, <blobAttribute>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
EXTRACT or REPLACE	(EXTRACT REPLACE)	EXTRACT says we want to extract the raster geometry into an attribute. REPLACE says we want to replace the raster geometry with what is in the attribute specified	No
<format>	(GEOTIFF TIFF BMP JPEG JPEG2000 GIF PNG)	The format the raster <blobAttribute> is written to or read from.	No
<formatAttribute>	String	The name of the attribute where the output raster format name is written. This should be identical to the selected <format> above.	No
<blobAttribute>	String	The name of the attribute where the binary raster data blob is to be written or read.	No

Configuration

This function does not accept configuration lines.

Description

The @RasterGeometry function either serializes the geometry of the feature into the Raster Blob Attribute based on the Writer Format or deserializes the Raster Blob Attribute and replaces the geometry of the feature.

When using EXTRACT, the raster geometry of the feature will be written using the selected <format> to a temporary file. The bytes of that file will then be copied in the <blobAttribute>. If a <formatAttribute> is specified, the name of the format will be written in that attribute.

When using REPLACE, the raster geometry of the feature will be replaced by what is in <blobAttribute>. The decoding is done according with the <format> specified. This function is useful if you want to read or write raster data in a database field.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a typical use of the @RasterGeometry function. In this example, the SamplingFactory is used to pass through every feature, while the @RasterGeometry function serializes the feature into the attribute _blobAttribute.

```

FACTORY_DEF * SamplingFactory                                \
    SAMPLE_RATE 1                                           \
    INPUT FEATURE_TYPE *                                    \
        @RasterGeometry(EXTRACT, GEOTIFF, _formatAttribute, \
        _blobAttribute)                                     \
    OUTPUT FEATURE_TYPE

```

In the next example, the serialized raster in the _blobAttribute is deserialized and replaces the raster geometry of the feature.

```

FACTORY_DEF * SamplingFactory                                \
    SAMPLE_RATE 1                                           \
    INPUT FEATURE_TYPE *                                    \
        @RasterGeometry(REPLACE, GEOTIFF, _blobAttribute)  \
    OUTPUT FEATURE_TYPE

```

@RasterNodata

```
@RasterNodata(<mode>[,<nodata_value>[,<replace_cell_values>]])
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<mode>	BAND PALETTE REMOVE	A string that specifies whether the function will operate on the band level or the palette level, or will remove the nodata value.	No
<nodata_value>	Reals or Integers	A numeric value specifying the nodata value to be set on the input raster. Optional if the <mode> is REMOVE.	Yes
<replace_cell_values>	ReplaceCellValues DoNotReplaceCellValues	The DoNotReplace-CellValues or Replace-CellValues options specifying tagging the nodata value and optionally replacing the cell values as well when <mode> is set to BAND. If this parameter is not specified, DoNotReplaceCellValues is the default.	Yes

Configuration

This function does not accept configuration lines.

Description

The @RasterNodata function is used to set or flag the nodata value on a raster's bands or palettes. The nodata value to be used should be specified in the <nodatavalue> parameter. The function operates on either the band level or the palette level depending on the value of the <mode> parameter.

In BAND mode, the function sets the nodata value and discards the original nodata value if it exists. If a band has a palette, the function will succeed in setting a new nodata value only if the value is of a key that already exists. In this mode, the function supports tagging the nodata value and optionally replacing the cell values as well.

In `PALETTE` mode, the function will succeed in setting the specified nodata value only if the input band(s) have at least one palette and a nodata key (on the band level) has already been set.

In `REMOVE` mode, the function will remove any nodata classification from the raster's bands and palettes. The `<nodata_value>` parameter is optional in this mode of operation.

Input features must contain raster geometries only.

This function currently does not support full color models or string interpretations.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example 1

The following example shows a typical use of the `@RasterNodata` function. `SamplingFactory` is used to filter the incoming features, while the `@RasterNodata` function sets the nodata on the palettes of the input raster to -99. This will produce a failure if the nodata key hasn't already been specified on the band.

```
FACTORY_DEF * SamplingFactory           \
    SAMPLE_RATE 1                       \
    INPUT FEATURE_TYPE *                \
        @RasterNodata (PALETTE, -99)    \
    OUTPUT FEATURE_TYPE *
```

Example 2

In this example, the `@RasterNodata` function tags the nodata on the bands of the input raster to 3. This will produce a failure if a band has a palette and doesn't have a key with the value 3.

```
FACTORY_DEF * SamplingFactory           \
    SAMPLE_RATE 1                       \
    INPUT FEATURE_TYPE *                \
        @RasterNodata (BAND, 3)         \
    OUTPUT FEATURE_TYPE *
```

Example 3

In this example, the `@RasterNodata` function removes the nodata value from all selected bands and palettes.

```
FACTORY_DEF * SamplingFactory           \
    SAMPLE_RATE 1                       \
```



```
INPUT FEATURE_TYPE *  
    @RasterNodata(REMOVE)  
OUTPUT FEATURE_TYPE *
```

\
\

@RasterProperties

@RasterProperties (<propertyiestype>)

Function Type: Attribute and Feature

Arguments:

Name	Range	Description	Optional
<propertyiestype>	RASTER BAND	A string that specifies which set of properties the caller wishes to obtain in the form of attributes.	No

Configuration

This function does not accept configuration lines.

Description

The @RasterProperties function is used to expose the properties of a raster input as attributes. The type of properties (raster, band, or palette) can be chosen by specifying RASTER or BAND for the <propertyiestype> parameter. Note that there are two sets of extents provided: 1.the _min and _max extents are the Minimum Bounding Rectangle (MBR) extents and may not match the values of the raster corners if the raster is rotated, and 2. the upper, lower, left and right extents that specify the exact corner locations of the raster even if the raster is rotated.

Specifying RASTER will add the following attributes to the feature:

```
_num_bands
_num_rows
_num_columns
_spacing_x
_spacing_y
_origin_x
_origin_y
_rotation
_min_x
_min_y
_max_x
_max_y
_cell_origin_x
_cell_origin_y
_upper_left_x
_upper_left_y
_upper_right_x
_upper_right_y
```

```

_lower_right_x
_lower_right_y
_lower_left_x
_lower_left_y

```

Specifying `BAND` will add the following band property attributes to a list with the name `_band{}`. Additionally, palette properties will be appended as attributes to a list named `palette{}`, which is appended to the `_band` list, which is appended to the feature:

```

_band{}.band_name
_band{}.band_interpretation
_band{}.band_bit_depth
_band{}.band_num_tile_rows
_band{}.band_num_tile_columns
_band{}.band_nodata
_band{}.band_num_palettes
_band{}.palette{}.palette_name
_band{}.palette{}.palette_key_interpretation
_band{}.palette{}.palette_value_interpretation
_band{}.palette{}.palette_bit_depth
_band{}.palette{}.palette_key_nodata
_band{}.palette{}.palette_value_nodata

```

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example 1

The following example shows a typical use of the `@RasterProperties` function. `SamplingFactory` is used to filter the incoming features, while the `@RasterProperties` function adds the raster properties as attributes on the feature.

```

FACTORY_DEF * SamplingFactory                                \
    SAMPLE_RATE 1                                           \
    INPUT FEATURE_TYPE *                                    \
        @RasterProperties(RASTER)                            \
    OUTPUT FEATURE_TYPE *

```

@Reformat

@Reformat (<direction> , <structId> , <attribute>)

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<direction>	DestEncode SrcDecode or DestDecode SrcEncode	Indicates the direction of the Reformat. DestEncodeSrcDecode means that the Reformat will write data from the fields defined by structId and store them in the attribute specified by <attribute> when it is invoked on a destination transfer specification line (or on a factory input or output clause). It will decode the data when it is specified on a source line. DestDecodeSrcEncode performs the opposite.	No
<structId>	Must Be Defined	Identifies the reformat structure definition to use for the Reformat function. These are defined on configuration lines.	No
<attribute>	Character String	This specifies the attribute into which the stored (encode operation) or retrieved (decode operation) is written.	No

Configuration

Reformat STRUCT_DEF <structId> [<fieldName> <fieldDef>] +

Name	Range	Description	Optional
<structId>	Character String	The name of the structure defined by the configuration line. This name is then referred to in @Reformat () statements to encode or decode data in the defined structure.	No

Name	Range	Description	Optional
<fieldName>	Character String	The name of the field in the structure being defined.	No
<fieldType>	Character String	The type of the field being defined. The field defined here can be any defined for the CAT type. See the <i>FME Readers and Writers</i> manual: <i>Relational Table</i> chapter, <i>Description of CAT Field Types</i> .	No

Description

This function is used to encode and decode data in attributes. One obvious use for this function is to pack and unpack attribute information into binary structures.

Before an encode operation can be performed, the target structure for the decoding or encoding must first be defined on a configuration line. There is no limit to the number of configuration lines that can be defined in a single mapping file.

When specified on a transfer specification destination line or on a factory input or output line, this function encodes data into the specified attribute when `DestEncodeSrcDecode` is specified. When `DestDecodeSrcEncode` is specified on the destination line or specification line of a factory, then `@Reformat()` decodes the value of the specified attribute and stores the results in attributes as defined in the structure definition.

The function also returns the encoded value.

Inverse Operation

When specified on the source line of a transfer specification, then the function encodes data into the specified attribute when `DestDecodeSrcEncode` is specified. When `DestEncodeSrcDecode` is specified on the source specification line then the `@Reformat()` decodes the value of the specified attribute and stores the result in attributes as defined by the configuration line.

The function also returns the encoded value in the inverse direction.

Example

In this example, the @Reformat function is used to store the definition for ellipse features into a binary structure.

The following configuration line defines the structure into which the ellipse parameters are stored. Note that the field names used in the structure are the field names of the values in the feature @Reformat is called on.

```
Reformat STRUCT_DEF EllipseStruct \
    PRIM_AXIS BigEndianReal(8,1) \
    SEC_AXIS BigEndianReal(8,9) \
    ROTATION BigEndianReal(8,17)
```

In the following lines, this definition is used to convert the parameters of the Design file arc into a single blob field to be stored in the SDE. When the source is IGDS, then the reverse occurs and the attribute named IGDS_INTERNAL is decoded to give the three attributes defined above in the structure.

```
IGDS * \
    igds_type igds_ellipse \
    igds_primary_axis %igds_primary_axis \
    igds_secondary_axis %igds_secondary_axis \
    igds_rotation %igds_rotation \
    igds_start_angle %igds_start_angle \
    igds_sweep_angle %igds_sweep_angle \

SDE30 FACILITYGRAP \
    sde30_type sde30_line \
    PRIM_AXIS %igds_primary_axis \
    SEC_AXIS %igds_secondary_axis \
    ROTATION %igds_rotation \
    STARTANGLE %igds_start_angle \
    SWEEPANGLE %igds_sweep_angle \
    @Arc(&PRIM_AXIS,&SEC_AXIS,45,&ROTATION) \
    @Reformat(DestEncodeSrcDecode, EllipseStruct, IGDS_INTERNAL)
```


@RemoveRasterBands

@RemoveRasterBands (<mode>)

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<mode>	REMOVE KEEP	This parameter specifies whether selected bands will be removed or kept (in which case all non-selected bands are removed).	No

Configuration

This function does not accept configuration lines.

Description

The @RemoveRasterBands function is used to remove unwanted bands from raster features.

The <mode> parameter specifies whether selected bands are removed or kept. When set to REMOVE, all the selected bands in the raster will be removed. When set to KEEP, the selected bands will remain while all the non-selected bands will be removed.

Input features must contain raster geometries only.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a typical use of the @RemoveRasterBands function. In this example, the SamplingFactory is used to pass through every feature. Then, the @SelectRaster function is used to select band 3; finally, the @RemoveRasterBands function removes that band.

```
FACTORY_DEF * SamplingFactory                                \  
    SAMPLE_RATE 1                                           \  
    INPUT FEATURE_TYPE *                                    \  
        @SelectRaster(3)                                     \  
        @RemoveRasterBands(REMOVE)                         \  
    OUTPUT FEATURE_TYPE *
```


@RemoveRasterPalettes

@RemoveRasterPalettes()

Function Type: Feature

Arguments: None

Configuration

This function does not accept configuration lines.

Description

The @RemoveRasterPalettes function is used to remove palette(s) from a raster. If the source band has no palettes, the raster will remain unchanged and the function will succeed.

This function supports raster band and palette selection.

Input features must contain raster geometries only.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a use of the @RemoveRasterPalettes function. In this example, the SamplingFactory is used to pass through every feature and any selected palettes will be removed. The features that have no selected palettes on any of their selected bands will not be affected.

```

FACTORY_DEF * SamplingFactory                                \
    SAMPLE_RATE 1                                           \
    INPUT FEATURE_TYPE *                                    \
        @RemoveRasterPalettes()                             \
    OUTPUT FEATURE_TYPE *
```



```
@ReinterpretRaster(RASTER, <Interpretation>, <Many-
ToOneBand>, <RGBAToRGB>, <ColorToColor>, <ColorToNu-
meric>, <NumericToColor>, <NumericToNumeric>,
<FloatToInteger>)
```

```
@ReinterpretRaster(BAND, <BandInterpretation>, <ColorTo-
Color>, <ColorToNumeric>, <NumericToColor>, <NumericToNu-
meric>, <FloatToInteger>)
```

```
@ReinterpretRaster(PALETTE, <PaletteInterpretation>,
<RGBAToRGB>, <ColorToColor>, <FloatToInteger>)
```

Function Type: Feature

Arguments:

186 • • *FME Functions and Factories* • •

Name	Range	Description	Optional
<ManyToOne>	(AVERAGE)	The action to take when converting from multiple bands to a single band.	No
<RGBAToRGB>	(DROPALPHA APPLYALPHA)	The action to take when converting from RGBA to RGB. DROPALPHA discards the alpha band and APPLYALPHA multiplies each RGB value with the corresponding, normalized alpha value.	No
<ColorToColor>	(CAST BOUNDED_CAST DATASCALE TYPESCALE)	The action to take when converting from one color interpretation to another. CAST performs a C-style cast on the values. BOUNDED_CAST performs a C-style cast on the values but also performs bounds-checking. DATASCALE scales the extremum of the source data values to the extremum of the destination data type. TYPESCALE scales the extremum of the source data type to the extremum of the destination type.	No
<ColorToNumeric>	(CAST BOUNDED_CAST DATASCALE TYPESCALE)	The action to take when converting from color interpretations to numeric interpretations. See <ColorToColor>.	No
<NumericToColor>	(CAST BOUNDED_CAST DATASCALE TYPESCALE)	The action to take when converting from numeric interpretations to color interpretations. See <ColorToColor>.	No
<NumericToNumeric>	(CAST BOUNDED_CAST DATASCALE TYPESCALE)	The action to take when converting from numeric interpretations to numeric interpretations. See <ColorToColor>.	No
<FloatToInteger>	(ROUND CEIL FLOOR)	The action to take when converting from floating-point values to integer. ROUND rounds the value to the nearest integer. CEIL gets the next integer which is greater than or equal to the floating-point value. FLOOR gets the next integer which is less than or equal to the floating-point value.	No

Configuration

This function does not accept configuration lines.

Description

The `@ReinterpretRaster` function is used to change the underlying interpretation of the bands and palettes of a raster. It can be used in three modes: `RASTER`, `BAND`, or `PALETTE`.

In `RASTER` mode, the function acts on the raster as a whole, ignoring selection, and can change the number of bands on the raster. For example, this mode can be used to convert a raster containing 3 bands directly to `RGB` or `RGBA`. The raster cannot contain any palettes in `RASTER` mode. Here is a summary of the possible conversions:

- Converting to `RGBA` is possible from one band (duplicating it and adding an opaque alpha band), three bands (adding an opaque alpha band), and four bands.
- Converting to `RGB` is possible from one band (duplicating it), three bands, and four bands if the four bands are already `RGBA`. Converting from four `RGBA` bands to `RGB` will either drop or apply alpha, depending on the chosen option. You can convert any four bands to `RGB` by first converting them to `RGBA`, then converting them to `RGB`.
- Converting to a single band is possible from any number of bands. This will average all input bands into the resulting band.

When averaging multiple bands, `DATASCALE` and `TYPESCALE` options are not permitted. It is possible to work around this limitation by doing the conversion in two steps. For example, to average a `RGB48` raster into a `GRAY8` band using `TYPESCALE`, consider first averaging to `GRAY16` in raster mode, and then using a `TYPESCALE` conversion to `GRAY8` in `BAND` mode.

In `BAND` mode, the function acts on each selected band individually. If the band contains palettes, regardless of selection, the palette keys will be modified to match the band if possible. This is only the case if the destination band interpretation is `UINT8`, `UINT16` or `UINT32`. If a raster contains multiple selected bands, each of these bands will be converted to the destination interpretation independently. Thus, if a `GRAY8` interpretation is requested on a three band raster with no palettes, and only the first two bands are selected, then only the first two bands are converted to `GRAY8`, the third band will be left untouched.

In `PALETTE` mode, the function acts on each selected palette in each selected band. This mode does not support selected bands without palettes. If a raster contains multiple selected bands with multiple selected palettes, each palette will be converted to the destination interpretation independently. Converting palettes to string interpretation will either create an integer string, or a CSV string, depending on the input interpretation. String palettes created using this function will currently have a maximum length of 63 characters.

Not all parameters passed to a function make sense all the time. For example, when converting to `INT16` in `BAND` mode, the `<ColorToColor>` and `<NumericToColor>` options will not be used because the destination type is not a color interpretation. Another example would be the `<FloatToInteger>` parameter when the destination interpretation is `REAL32` or `REAL64`. Also, some parameters might be used in some cases, but not others, like the `<RGBAToRGB>` parameter, which is going to be used only when the input palette or raster has a `RGBA` interpretation. All parameters always have to be provided to define the behavior of the function for all possible input interpretations.

The `TYPESCALE` option is mainly used when converting color interpretations to color interpretations, so that both the source and destination look exactly the same when rendered in the Viewer. The `DATASCALE` option is mostly used when converting numeric interpretations to color interpretations, in order to give a good visualization of the data in the Viewer.

In `BAND` mode, the function will not copy data if it does not have to. When converting between interpretations that have the same underlying data type and without asking for a `DATASCALE`, no superfluous copy will occur. This type of conversion is very efficient.

Sequential functions are not combined. If several `ReinterpretRaster` functions are performed in sequence, or if the resulting converted raster is output to a format that cannot support the converted type, then potentially multiple conversions will take place and data quality and translation performance may suffer.

Conversion of the datatypes of raster bands may cause existing data and no data values to be combined. For example, converting the data values from `UInt32` to `UInt8` may cause several possible source values to be converted to the same destination value. If one of the values was no data then it is possible that several data and no data values may share the same destination value and thus the data values will be destroyed. This can be mitigated by either removing the no data value before conversion or setting the no data value intelligently if the data values and conversion operation are well understood.

This function supports sub-selection of raster bands and palettes in `BAND` and `PALETTE` mode only.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a use of the `@ReinterpretRaster` function. In this example, the `SamplingFactory` is used to filter the incoming features, and assuming rasters with palettes are input then, the `@ReinterpretRaster`

function converts each selected palette on each selected band of each input raster an RGBA (red, green, blue, alpha) interpretation. The `DROPALPHA` and `CEIL` parameters will not be used during this conversion, because we are converting to `RGBA64`. The `TYPESCALE` parameter means that the source and destination values will be proportional.

```
FACTORY_DEF * SamplingFactory \
    SAMPLE_RATE 1 \
    INPUT FEATURE_TYPE * \
        @ReinterpretRaster (PALETTE, RGBA64, DROPALPHA, TYPESCALE, \
    CEIL) \
    OUTPUT FEATURE_TYPE *
```

Here, the data type is set to `UINT8` in `BAND` mode. This function will attempt to convert every selected band in the input raster into a band of 8 bit values, casting with bounds from color interpretations, scaling by data values from numeric interpretations and rounding from floating-point values:

```
FACTORY_DEF * SamplingFactory \
    SAMPLE_RATE 1 \
    INPUT FEATURE_TYPE * \
        @ReinterpretRaster (BAND, UINT8, CAST, BOUNDED_CAST, CAST, D \
    ATASCALE, ROUND) \
    OUTPUT FEATURE_TYPE *
```

In this last example, the function is used in `RASTER` mode to convert a whole raster to the `RGB48` interpretation. Assuming rasters without palettes are input, this function will attempt to convert all bands in the input raster into `RED16`, `GREEN16`, and `BLUE16` bands, applying the alpha band, scaling by data type from color interpretations, and casting values from numeric interpretations. If a raster containing four bands that are not `RGBA` is input, an error will occur, and multiple conversions steps are required.

```
FACTORY_DEF * SamplingFactory \
    SAMPLE_RATE 1 \
    INPUT FEATURE_TYPE * \
        @ReinterpretRaster (RASTER, RGB48, AVERAGE, APPLYALPHA, TYP \
    ESCALE, CAST, CAST, CAST, ROUND) \
    OUTPUT FEATURE_TYPE *
```


@ReplaceRasterCellValues

Note: This function has been renamed from `RasterCellValueReplacer`.

Function Type: Feature

Name	Range	Description	Optional
<greater than value>	Real64	The lower bound of a range of values to replace. All values in the source raster greater than or equal to this value will be replaced by the new value. This value will be scaled to a valid value in the data range of the source raster. At least one of the greater than value or the less than value must be specified.	Yes
<less than value>	Real64	The upper bound of a range of values to replace. All values in the source raster less than or equal to this value will be replaced by the new value. This value will be scaled to a valid value in the data range of the source raster. At least one of the greater than value or the less than value must be specified.	Yes
<new value>	Real64	The value with which to replace the specified range of values.	No
<replace nodata>	ReplaceNodata DoNotReplaceNodata	The ReplaceNodata and DoNotReplaceNodata options specify whether nodata values will be replaced. If ReplaceNodata is selected, nodata values will be replaced just as any other values. If DoNotReplaceNodata is selected, then nodata values will not be replaced, even if they fall within the specified bounds. If this parameter is not specified, ReplaceNodata is the default.	Yes

Configuration

This function does not accept configuration lines.

Description

This function is used to replace values in a raster with new values. The upper and lower bounds can specify either a range of values within the raster to replace or a single value to replace.

To replace all values less than a given value, simply specify the `<less than value>` parameter without specifying the `<greater than value>` parameter, and provide a new value with which to replace that range.

To replace all values greater than a given value, simply specify the `<greater than value>` parameter without specifying the `<less than value>` parameter, and again provide a new value with which to replace the range.

To replace a single value, simply provide identical values for the `<greater than value>` and `<less than value>` parameters.

To replace all values except for a single value, you must specify a `<greater than value>` exceeding the value you do not want to be replaced, and a `<less than value>` smaller than the value you do not want to be replaced. These values may vary depending on the precision of the source raster's data type.

Parameter values will be scaled appropriately to the source raster's data type, and set to the largest or smallest possible value for that data type in cases where they are too large or small.

Rasters that contain bands with palettes are not supported at this time.

This function supports sub selection of raster bands.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a use of the `@ReplaceRasterCellValues` function. In this example, the `SamplingFactory` is used to filter the incoming features, while the `@ReplaceRasterCellValues` function specifies that all occurrences of the value "100" should be replaced with "0":

```
FACTORY_DEF * SamplingFactory \
    SAMPLE_RATE 1 \
    INPUT FEATURE_TYPE * \
    @ReplaceRasterCellValues(100, 100, 0, ReplaceNodata) \
```

OUTPUT FEATURE_TYPE *

In this case, a range of values is specified to be replaced. Furthermore, any nodata values that fall within this range will not be replaced:

```
FACTORY_DEF * SamplingFactory \
SAMPLE_RATE 1 \
INPUT FEATURE_TYPE * \
    @ReplaceRasterCellValues(0, 100, -999, \
DoNotReplaceNodata) \
    OUTPUT FEATURE_TYPE *
```

@Relate

```
@Relate(<relationId>, (Read|Write))
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<relationId>	Must Be Defined	Identifies the relation definition to execute. Relations are defined by <code>Relate RELATION_DEF</code> statements.	No
<direction>	Read or Write	Indicates the direction of the relation. Read means that @Relate will read data from the relational tables <i>into</i> the feature when it is invoked on a destination transfer specification line (or on a factory input or output clause). It will write data when it is invoked on a source line. Note: In previous FME versions, this was called <code>DestReadSrcWrite</code> . Write means the opposite. Note: In previous FME versions, this was called <code>DestWriteSrcRead</code> .	No

Description

The @Relate function combines relational data held in auxiliary databases with FME features. The function can be configured to perform simple single-table joins, or complex, multi-table, multi-record joins. The relational data may be held in one or more of the supported file formats, or in a live database.

The @Relate function is a feature function that *adds* data from a relational table to the FME feature when it is *reading*. When it is *writing*, it *extracts* data from the FME feature and writes it to a relational table.

This function requires considerable configuration before it may be invoked.

- The location of the table must be specified using the `TABLE_LOCATION` configuration option.
- The structure of each table used by the relation definition must be specified using the `TABLE_DEF` configuration option.
- The relation itself must be defined. The relation operates on a logical table name and is independent of the actual physical structure of the table. The relation is defined using the `RELATION_DEF` configuration option.

Relations may be one-to-one (1:1), in which case a single record is extracted from the table and added to the FME feature when the relation is executed in the forward direction. If there is no matching record or there is more than one matching record, then the translation will be terminated. A one-to-zero or one (1:0..1) relation is similar to a 1:1 relation, but when no matching database record is found, no action is taken. If more than one matching record is found then the translation is terminated. A one-to-zero or one plus (1:0..1+) relation may match to multiple records in the database, but only attributes from the first matching record are extracted. No error is returned if no matching record or more than one matching records found. One-to-many (1:M) relations add 0 or more table records to the FME feature. These records may be sorted before being added to the FME feature, and for debugging purposes they may be logged. The table field and feature attribute names specified in the `JOIN` clause of the relation definition select the record(s) from the table. The FME feature must have previously had values for any attributes named in a `JOIN` clause. Once a record is selected, the `TRANSFER` clause moves the table fields into the named FME feature fields. Once this is done, the feature functions, if any, attached to the relation are executed. The most common feature function used is another `@Relate` call, to allow the relation to pull information from several tables in the FME feature.

Attribute value functions can be used in place of table field and feature attribute names in `JOIN` and `TRANSFER` clauses. This allows attribute value functions to be used to calculate field values. The values of table fields and feature attributes can be passed to commands by putting the value-of operation (an ampersand `&`) in front of their names. Depending on their placement and the direction of `@Relate`, the attribute functions operate according to the following rules:

- When the relation is executed in the forward direction, the left (table) side of the `TRANSFER` clause is the *source* of attribute values. Any attribute functions on this side are executed in the *forward* direction. The value transferred to the feature side of the clause is the result of the inverse execution of the function.
- The right (feature) side of the `TRANSFER` clause is the destination of the table values. Any attribute functions on this side are executed in the *inverse* direction. The value stored in the feature attribute is the result of the forward execution of the function.
- The `JOIN` clause works in the opposite way when `@Relate` is executed in the forward direction. This is because the `JOIN` looks in the table for the values specified by the feature. In this case, then, the right-hand (feature) side of the `JOIN` clause is the *source* of the key values. Any attribute functions on this side are executed in the *inverse* direction. The resulting value is passed as the key value to the left-side (table side) of the clause.
- The left (table) side of the `JOIN` clause is the destination of the key values. Any attribute functions on this side are executed in the *forward* direction, since functions on the destination side always execute in this way. The value computed is then searched for in the table to identify the record to be

- extracted.
- When the relation is 1:M, several rows may be pulled from the relational table. The `TRANSFER` clauses and feature functions of the relation are executed for each row. Any attribute list field names on the feature side are expanded to include the row number in their `{}`. Rows are numbered starting at 0.

Tip

Use the `Relate` option on `FME_DEBUG` to log all the rows read by a query.

A cache is used to improve efficiency when repeatedly fetching the same rows from the database. This cache is used automatically when reading rows, but may also be prepopulated by an SQL query of the database, called a *prefetch query*. By using a prefetch query to retrieve a number of database records in a single query, performance will be improved by avoiding database round-trip queries. See the `PREFETCH_QUERY` in the configuration section below.

Inverse Operation

When a relation is executed in the inverse direction, information is extracted from the FME feature and output to the table. In the case of a 1:1 relation, a single record is output to the table, while 1:M relations add 0 or more records to the table.

The inverse execution makes use of the field combinations listed in the `UNIQUE` clause to prevent writing duplicate records to the output tables. For some tables and relation combinations, this is not an issue. In these cases, the `NOTUNIQUE` keyword is used in place of the `UNIQUE` clause.

In the inverse case, the FME feature is the source for all attribute data. Values are transferred from the right-hand (feature) side of both the `JOIN` and `TRANSFER` clauses to the table field names. Once a candidate table record has been constructed, a check is made to see if a record is already present in the table that is identical according to the fields listed in the `UNIQUE` clause and, if so, the candidate record is discarded. Otherwise, the record is written to the table. Once the record is written, any feature functions that were part of the relation are executed in the inverse direction.

When the relation is 1:M, several rows may be output to the relational table. The `JOIN` and `TRANSFER` clauses and feature functions of the relation are executed in the inverse direction for each row. Any attribute list field names on the feature side are expanded to include the row number in their `{}`. Rows are numbered starting at 0. When no data for a row can be found in the feature, then the `@Relate` function terminates.

Attribute value functions present in the `JOIN` and `TRANSFER` clauses are executed according to the following rules:

- When the relation is executed in the inverse direction, the right-hand (feature) side of the `JOIN` and `TRANSFER` clauses is the *source* of attribute values. Any attribute functions on this side are executed in the *inverse* direction, since functions on the source side are always executed backwards. The value transferred to the table side of the clause is the result of the inverse execution of the function.
- The left-hand (table) side of the `JOIN` and `TRANSFER` clauses is the destination of the feature values. Any attribute functions on this side are executed in the *forward* direction, since functions on the destination side always executed in this way. The value stored in the table is the result of the forward execution of the function.

Configuration

```
Relate RELATION_DEF <relationId> <cardinal>          \
TABLE <tableId>                                     \
[UNIQUE(<tableField>[,<tableField>]+) |              \
NOTUNIQUE]                                           \
[JOIN      <tableField> TO <featureField>]+          \
[SORT_BY  [<tableField> (NUMERIC|ALPHA)]+ ]          \
[SORT_DIRECTION (ASCENDING|DESCENDING)]            \
[MATCHED_RECORDS_ATTR <countAttr>]                 \
[LOG_ROWS]                                           \
[TRANSFER <tableField> TO <featureField>]*           \
[TRIM_TRAILING (YES|NO)]*                           \
[<featFunc>] *
```

Tip

The `JOIN` clause identifies the key fields that must match in both the FME feature and the table row. The `TRANSFER` clause identifies fields which should be copied between the FME feature and the table row. If no `TRANSFER` clauses are specified, then all the fields are copied from the table row to the feature.

Name	Range	Description	Optional
<relationId>	Character String	The name of the relation defined by the configuration statement. This name is used as the argument when the @Relate function is called.	No
<cardinal>	1:0..1 1:0..1+ 1:1 1:M	Defines the cardinality of the relation. 1:0..1 indicates that there is either 0 or 1 row in the table for each feature. 1:0..1+ indicates that there may be 0 or more matching rows but only the first matched row will be extracted. 1:1 implies that there is exactly 1 row in the table for each feature. 1:M implies that there is 0 or more rows in the table for each feature. 1:M relations make use of <i>feature attribute lists</i> to store the attribute data in FME features. When a 1:M relation is used, the optional SORT_BY and SORT_DIRECTION clauses specify the order that the rows will be added to the attribute list.	No
<tableId>	Character String	The logical name of the table used by this relation. The table's structure and location are defined by TABLE_LOCATION and TABLE_DEF statements.	No
<tableField>	The name of any field in the table, or an attribute value function.	<p>The UNIQUE clause requires a list of table field names. The values of these fields are considered to form a unique key that the FME uses when outputting to the table. This is used by some 1:1 relations to avoid writing duplicate records to tables. 1:M relations by definition are NOTUNIQUE.</p> <p>Table field names are used in JOIN and TRANSFER clauses to identify the fields in the table that are joined or transferred to the corresponding fields in the FME feature. As well, in these clauses an attribute function may be used in place of the table field name.</p> <p>The join between attribute values and table fields takes place after all trailing space characters have been removed. This sometimes requires a TRIM operation to be performed on the database side, which can defeat the database indexes in some cases. If it is known that the database fields contain no trailing spaces, specifying TRIM_TRAILING NO on the relation definition may provide a performance boost.</p>	No

Name	Range	Description	Optional
<featureField>	The name of any FME feature attribute, or an attribute value function.	An FME feature attribute name. If the relation is 1:M, and the feature attribute name contains an empty set of parenthesis {}, the parenthesis will be filled with the current matching row number. Rows are numbered starting at 0. In JOIN and TRANSFER clauses, an attribute function may be used in place of the feature attribute name.	No
<featFunc>	Any Valid ^a Feature Function	One or more feature functions may be specified at the end of a relation definition. These functions are called after the JOIN and TRANSFER processing is complete. If the relation is being executed in the forward direction, the functions will be called in the forward direction. If the relation is executed in the inverse direction, then the inverse of these functions will be called. If the relation is 1:M, then these functions will be called <i>once for each row</i> that is processed.	No
<countAttr>	Any valid attribute name	Specifies the name of an attribute that will receive the number of database records that the feature matched. This value can be used in a 1:0..1 relation to determine whether a feature matched a record in the database, or in 1:0..1+ or 1:M relation to determine how many database records a feature matched.	Yes

- a. Any feature function can be called, including other @Relate functions. This allows relations to cascade across several tables.

```
Relate TABLE_DEF <tableId> <tableType>           \  
  [DATABASE_SERVER_TYPE <serverType>]             \  
  [DATABASE_USER_NAME <userName>]                 \  
  [DATABASE_PASSWORD <password>]                   \  
  [DATABASE_NAME <dbName>]                         \  
  [DATABASE_USE_SSPI <useSspi>]                     \  
  [DATABASE_PORT <port>]                           \  
  [DATABASE_VERSION <version>]                     \  
  [<fieldName> <fieldType>]+
```

Name	Range	Description	Optional
<tableId>	Character String	The logical name of the table being defined. This name is used in <code>RELATION_DEF</code> and <code>TABLE_LOCATION</code> statements.	No
<tableType>	ASCII CAT CSV DBF DATABASE	The type of the table being defined. ASCII : Repeating free form ASCII CAT : Column-Aligned Text (FORTRAN-style file) CSV : Comma-separated value file DBF : dBASE III file DATABASE : Connection to a live database For reader keywords, see the <i>CSV</i> , <i>DBF</i> and <i>Database</i> chapters of the <i>FME Readers and Writers</i> manual.	No
<serverType>	ODBC ORACLE ORACLE8 MDB POSTGRES POSTGIS DB2 EXCEL MSSQL_ADO SDE30	The type of server being used.	Required for connections to a live database.
<userName>	Character String	The username that accesses the database. Some ODBC sources ignore this value.	Required for connections to a live database unless <code>useSspi</code> is set to YES.
<password>	Character String	The password that accesses the database.	Required for connections to a live database unless <code>useSspi</code> is set to YES.
<useSspi>	YES NO	To use Windows Authentication instead of username and password with SQL Server, set this parameter to YES.	Yes

Name	Range	Description	Optional
<dbName>	Character String	This is the name of the database. ODBC, ORACLE, and ORACLE8 databases ignore this setting. For ArcSDE, this is the ArcSDE dataset.	Required for connections to a live database.
<port>	Character String	This is the TCP/IP port for access to a remote PostGIS/PostgreSQL database. ODBC, ORACLE, and ORACLE8 and many other databases ignore this setting. For ArcSDE, this is the ArcSDE instance.	Required for connections to a live database.
<version>	Character String	For ArcSDE, this specifies the ArcSDE version. See the VERSION_NAME directive in the ESRI ArcSDE Reader/Writer documentation for more details.	Yes
<fieldName>	Character String	The name of the field. For DBF files, the field names must be in uppercase and 10 characters or less in length. Certain table types may reserve specific field names for special purposes.	No
<fieldType>	See table below.	The type of the field. The allowable field types depend on the type of database file. If the tableType is ASCII, CAT, CSV, or DBF, refer to the <i>Relational Table Reader/Writer</i> in the <i>FME Readers and Writers</i> manual for the allowable field types. If the tableType is DATABASE, refer to the table below. Certain table types may reserve specific field types for special purposes.	No

When the tableType is DATABASE, the permissible values for fieldType are:

Field Type	Description
number (<width>, <decimals>)	Number fields store floating point values. The width parameter is the total number of characters allocated to the field, including the decimal point. The decimals parameter controls the precision of the data and is the number of digits to the right of the decimal.
char (<width>)	Character fields store fixed length strings. The width parameter specifies the number of characters that can be stored. When a character field is written, it is right-padded with blanks, or truncated, to fit the width. When a character field is retrieved, any padding blank characters are stripped.

Field Type	Description
logical	Logical fields store TRUE/FALSE data. Data read or written from/to such fields must always have a value of either true or false.
date	Date fields store dates as character strings with the format YYYYMMDD.

Relate TABLE_LOCATION <tableId> <pathname>

Name	Range	Description	Optional
<tableId>	Character String	The logical name of the table whose location is being set. The table's structure is defined by a TABLE_DEF statement. For Open DataBase Connectivity (ODBC) data sources, this tableId is also the name of the table in the database.	No
<pathname>	Character String	The location of the table. If the tableType specified in the corresponding TABLE_DEF statement is not DATABASE, this field specifies the table location on the file system. If the tableType is DATABASE and the DATABASE_SERVER_TYPE is ODBC, this field specifies the ODBC data source name. If the DATABASE_SERVER_TYPE is ORACLE or ORACLE8, then it is the SQL*Net service name of the database. If the DATABASE_SERVER_TYPE is MDB or EXCEL, then it is the location of the database file on the file system. For ArcSDE, this is the server name. See Example 4 for details on using @Relate with live databases.	No

Relate CACHE_SIZE <tableId> <size>

Name	Range	Description	Optional
<size>	Integer	Specifies the size for the local query result cache, in rows. Results returned from the database are stored locally in order to speed up queries against remote databases. The default cache size is 5000. If the number of rows that will be joined is known before translation, this number may be changed to match it. This parameter is only available if tableType is DATABASE.	Yes

Relate PREFETCH_QUERY <tableId> <SQL query>

Name	Range	Description	Optional
<SQL query>	Any valid fully formed SQL query, which may include references to FME feature attributes via the "value of" (&) operator.	Specifies a query of the database, the results of which will be added to the cache of database information. The cache is intended to speed up subsequent access of the database during the read operations required to perform a join. By default, no prefetch query will be performed. In addition, the prefetch query will be ignored if the tableType is not Database.	Yes

The PREFETCH_QUERY specifies an SQL SELECT statement which will be used to query the database for rows that will likely be requested later. The results of the query will be added to the database cache, which will reduce the number of individual queries placed against the database as features request their corresponding rows.

Because performing a database query round-trip can be expensive, using this mechanism to preload a number of result rows into the local database cache can result in substantial performance improvements.

If the number of rows in the database is relatively few, then a query like:

```
SELECT * from TABLENAME
```

would result in all the records from the database being extracted, in a single query, and held locally. Then as features have @Relate executed on them, no visit to the database is needed to satisfy the query.

If the number of records in the table is excessive, it is not practical to cache them all locally during a translation run. In this case, a subset of the records can be extracted by adding a `WHERE` clause to the `SELECT` statement:

```
SELECT * from TABLENAME where BASENAME = '92b034'
```

In this case, it is known that for the entire run, all the features will be related to rows in the database where the `BASENAME` column had the value `92b034`, and so only these records should be held in the local cache.

The prefetch query can include references to feature attributes. This is typically used to preload the cache with rows likely related to a set of features who share a common attribute. Whenever the attributes referenced in the prefetch query change, the cache is cleared and the prefetch query will be re-executed. Therefore, it is important that when this facility is used, the features arrive in an order that minimizes the changes in their values. The FME's 'value of' operator (&) is used to de-reference feature attribute values in the query. For example, a prefetch query like:

```
SELECT * from TABLENAME where BASENAME = '&igds_basename'
```

would initially cache all the records from the table whose `BASENAME` column had a value matching the value held in the `igds_basename` attribute of the first feature encountered by @Relate. It would also cause the cache to be reloaded each time the `igds_basename` attribute changed from one feature to the next.

A note about prefetch queries:

Normally, a prefetch query exists to satisfy potential database matches; a request to the database will still be required if a match is not found in the prefetch cache. It is also possible to mark a prefetch query as “exhaustive”, which means that it describes all possible matches; in this case, a failure to form a match in the resulting cache means that there is no possible match in the database, and thus a round trip to the database is avoided.

Any query may be marked as being exhaustive by prepending a “plus” sign to the query. (e.g. “+SELECT * from TABLENAME where BASENAME = '&igds_basename'”) Additionally, queries of the form “SELECT * FROM <tableName>” are always assumed to be exhaustive.

Example 1

A Design file contains a set of shapes on level 45. Each shape has its graphic group set to a unique number, which can be used as a key to an associated CSV file containing information on the forest species and logging company assigned to the polygon. This is translated into a single Shape file, which holds these attributes in the element's attribute table.

```
# The definition of the Shape file.
```


The results of the translation are shown below.

```
1001,fir,MACBLO
1002,spruce,CANFOR
```

POLYID	SPECIES	COMPANY	AREA
1001	fir	MACBLO	500.201
1002	spruce	CANFOR	201.405

Example 2

Building on the previous example, an additional DBF file has been located. This file maps the 6-character company code to the company's full name. Shape file users would prefer to see the long name in their data rather than the code. The following mapping file accommodates this.

```

      SHAPE_DEF forestpoly      SHAPE_GEOMETRY shape_polygon      \
                                POLYID        number(4,0)         \
                                SPECIES        char(15)            \
                                COMPNAME       char(40)            \
                                AREA           number(10,3)        \
#-----
      Relate TABLE_LOCATION forestTable /usr/data/forest.csv

# The location of the new DBF file.
      Relate TABLE_LOCATION company /usr/data/company.dbf
      Relate TABLE_DEF      forestTable CSV                      \
                                polygonID        number(4,0)       \
                                forestSpecies    char(15)          \
                                companyId        char(6)           \
# The definition of the new DBF file.
      Relate TABLE_DEF company DBF                              \
                                COMPID          char(6)            \
                                FULLNAME       char(40)           \
      Relate RELATION_DEF CompanyID_To_Name 1:1                  \
                                TABLE company                    \
                                UNIQUE(COMPID)                    \
                                JOIN COMPID TO compId              \
                                TRANSFER FULLNAME TO COMPNAME      \
      Relate RELATION_DEF Poly_To_Forest 1:1                      \
                                TABLE forestTable                \
                                UNIQUE(polygonID)                 \
                                JOIN polygonID TO POLYID          \
                                TRANSFER forestSpecies TO SPECIES \
                                TRANSFER companyId TO compId      \
                                @Relate(CompanyID_To_Name)        \
      IGDS 45 idgs_type idgs_shape                                \
                                idgs_graphic_group %pid           \
      SHAPE forestpoly POLYID %pid                                \
                                AREA @Area()                      \
                                @Relate(Poly_To_Forest,Read)

```

The @Relate clause goes to the company table and fetches out the company's full name. Note that since the Poly_To_Forest relation is 1-1, the @Relate could have been placed on the Shape line instead of here.

If the original Design file had exactly 2 polygons with graphic group numbers 1001 and 1002, the original forestTable CSV file contained:

```

1001,fir,MACBLO
1002,spruce,CANFOR

```

and the original company DBF file contained:

COMPID	FULLNAME
MACBLO	MacMillan Bloedel
CANFOR	Canada Forest Products

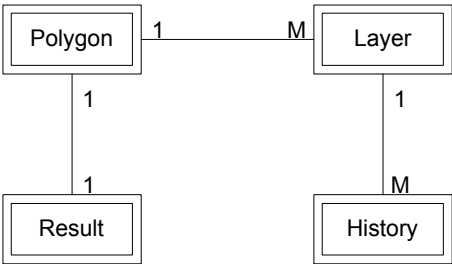
the resulting Shape file would contain two polygons, and its attribute table would look like this:

POLYID	SPECIES	COMPNAME	AREA
1001	fir	MacMillan Bloedel	500.201
1002	spruce	Canada Forest Products	201.405

Example 3

This example illustrates the use of the @Relate command to join forestry data from a Design file and several related attribute tables into a single SAIF object. Several one-to-many relationships are traversed to accomplish the join. The same mapping file can be used to go both to and from SAIF.

To assist in understanding the example, this entity relationship diagram sketches the relationships between each of the four auxiliary data tables:



The Design data file uses the text node number field to store a polygon number. This number is used as the key to the Polygon table, which in turn can be related to the other tables.

The SAIF object model for this data is shown below, using the Class Syntax Notation language:

```
<AbstractObject
  subclass: TreeStuff::SAFE
  attributes: saif_treeCode String
    // Note that the treecode could be made into an enum
```

```

        saif_percentage Real
    >

```

The HistoryStuff and LayerStuff AbstractObjects correspond to the History and Layer tables.

```

<AbstractObject
  subclass: HistoryStuff::SAFE
  attributes:  saif_actYear1 Integer
               saif_actYear2 Integer
               saif_activityCode String
               // Of course, activityCode could also be an enum
>
<AbstractObject
  subclass: LayerStuff::SAFE
  attributes:  saif_layerNumber Integer
               saif_rank Integer
               saif_trees List(TreeStuff::SAFE)
               saif_history List(HistoryStuff::SAFE)
>
<Enumeration
  subclass: InOpCode::SAFE
  values: go nogo
>
<GeographicObject
  subclass: ForestStand::SAFE
  attributes:  saif_id Integer
               saif_mapsheetId String
               saif_inopCode InOpCode::SAFE
               saif_tsaNum Integer
               saif_tsbNum Integer
               saif_fizCode String
               saif_layerStuff List(LayerStuff::SAFE)
>
<SpatialDataSet
  subclass: ForestStandComposite::SAFE
  restricted:  geoComponents{}: ForestStand::SAFE
>

```

The relevant portion of the mapping file, which joins the Design file spatial data to the tabular data to produce SAIF objects, is shown below.

Notice the use of macros to provide a single definition of the directory holding the input files.

```

MACRO baseDir /usr/safe/forestData
MACRO MAPSHEETID 92b034

Relate TABLE_LOCATION polygon $(baseDir)/polygon.csv
Relate TABLE_LOCATION history $(baseDir)/history.csv
Relate TABLE_LOCATION layer $(baseDir)/layer.csv
Relate TABLE_LOCATION result $(baseDir)/result.csv

# -----

```

```

Relate TABLE_DEF polygon CSV \
    mapsheetId char(10) \
    polygonId number(3,0) \
    inopCode char(2) \
    layerCount number(3,0) \
    historyCount number(3,0)

Relate TABLE_DEF layer CSV \
    mapsheetId char(10) \
    polygonId number(3,0) \
    layerNum number(3,0) \
    rank number(3,0) \
    tree1Code char(2) \
    tree1Percent number(6,3) \
    tree2Code char(2) \
    tree2Percent number(6,3)

Relate TABLE_DEF history CSV \
    mapsheetId char(10) \
    polygonId number(3,0) \
    layerNum number(3,0) \
    activityYear1 number(4,0) \
    activityYear2 number(4,0) \
    activity string

Relate TABLE_DEF result CSV \
    mapsheetId char(10) \
    polygonId number(3,0) \
    tsaNum number(3,0) \
    tsbNum number(3,0) \
    fizCode char(4)

# The lookup table is used to convert codes used in the original
# file to SAIF codes.

Lookup InOpCodeLUT AA go BB nogo

Relate RELATION_DEF Id_To_Polygon 1:1 \
    TABLE polygon \
    UNIQUE(polygonId,mapsheetId) \
    JOIN polygonId TO saif_id \
    JOIN mapsheetId TO saif_mapsheetId \
    TRANSFER layerCount TO @NumElements(saif_layerStuff) \
    TRANSFER historyCount TO \
        @NumElements(saif_layerStuff{}.saif_history) \
    TRANSFER @Lookup(InOpCodeLUT,&inopCode) TO saif_inopCode \
    @Relate(Polygon_To_Resultant,Read) \
    @Relate(Polygon_To_Layer,Read)

# The Id_To_Polygon relation is the primary relation. After it
# is done transferring values, it invokes both the
# Polygon_To_Resultant and Polygon_To_Layer relations.

```

```

Relate RELATION_DEF Polygon_To_Resultant 1:1
TABLE result
UNIQUE(polygonId,mapsheetId)
JOIN      polygonId      TO saif_id
JOIN      mapsheetId     TO saif_mapsheetId
TRANSFER tsaNum          TO saif_tsaNum
TRANSFER tsbNum          TO saif_tsbNum
TRANSFER fizCode         TO saif_fizCode

# This relation is 1 to many. It expects to find several rows in
# the table which match the polygonId and mapsheet id. For each
# row that matches, the {} are replaced by the row number and
# the values are transferred across. As well, for each row that
# matches, the Layer_To_History relation is invoked.

Relate RELATION_DEF Polygon_To_Layer 1:M
TABLE layer
NOTUNIQUE
JOIN polygonId TO saif_id
JOIN mapsheetId TO saif_mapsheetId
TRANSFER layerNum TO saif_layerStuff{}.saif_layerNumber
TRANSFER rank TO saif_layerStuff{}.saif_rank
TRANSFER tree1Code TO saif_layerStuff{}.saif_trees{0}
      .saif_treeCode
TRANSFER tree1Percent TO saif_layerStuff{}.saif_trees{0}
      .saif_percentage
TRANSFER tree2Code TO saif_layerStuff{}.saif_trees{1}
      .saif_treeCode
TRANSFER tree2Percent TO saif_layerStuff{}.saif_trees{1}
      .saif_percentage
@Relate(Layer_To_History,Read)

Relate RELATION_DEF Layer_To_History 1:M
TABLE history
NOTUNIQUE
JOIN polygonId TO saif_id
JOIN mapsheetId TO saif_mapsheetId
JOIN layerNum TO saif_layerStuff{}.saif_layerNumber
TRANSFER activityYear1 TO saif_layerStuff{}.saif_history{}
      .saif_actYear1
TRANSFER activityYear2 TO saif_layerStuff{}.saif_history{}
      .saif_actYear2
TRANSFER activity      TO saif_layerStuff{}.saif_history{}
      .saif_activityCode

# -----

# Finally the transfer spec.
IGDS 14 igds_type igds_text_node igds_node_number %nodenum
SAIF ForestStand::SAFE position.geometry.Class Point
      saif_id %nodenum
      saif_mapsheetId $(MAPSHEETID)

```

```
@Relate(Id_To_Polygon,Read) @Log
```

Notice that the primary relation is activated on the SAIF line. When SAIF is the output format, then @Relate will go and read data into the SAIF object from the tables. When SAIF is the source format, @Relate will write data out to the tables.

The tables below show the original source data. A listing of the FME object that is ultimately written out to SAIF for forest polygon 302 follows.

Polygon Table

mapsheetID	polygonID	inopCode	layerCount	historyCount
92b034	302	AA	2	5
92b034	303	BB	3	4

Layer Table

mapsheet Id	polygon Id	layer Num	rank	tree1 Code	tree1 Percent	tree2 Code	tree2 Percent
92b034	302	1	1	FIR	78	SPRUCE	22
92b034	302	2	0	SPRUCE	60	WILLOW	78
92b034	303	1	4	FIR	90	PINE	10
92b034	303	2	1	SPRUCE	65	FIR	35
92b034	303	3	8	PINE	70	WILLOW	30

History Table

mapsheetId	polygonId	layerNum	activity Year1	activity Year2	activity
92b034	302	1	1990	1991	HARVEST
92b034	302	1	1992	1993	GROOM
92b034	302	1	1993	1994	HARVEST
92b034	302	2	1993	1994	HARVEST
92b034	302	2	1991	1993	GROOM
92b034	303	1	1993	1994	HARVEST
92b034	303	1	1990	1993	HARVEST
92b034	303	2	1990	1994	GROOM
92b034	303	3	1990	1994	GROOM

Result Table

mapsheetId	polygonId	tsaNum	tsbNum	fizCode
92b034	302	30	4	BUBBLY
92b034	303	21	55	FLAT

Resulting FME Object (id=302)



Feature Type: ForestStand::SAFE

Attribute	Value
saif_id	302
saif_mapsheetId	92b034
saif_inopCode	go
saif_tsaNum	30
saif_tsbNum	4
saif_fizCode	BUBBLY
saif_layerStuff{0}.saif_layerNumber	1
saif_layerStuff{0}.saif_rank	1
saif_layerStuff{0}.saif_trees{0}.saif_treeCode	FIR
saif_layerStuff{0}.saif_trees{0}.saif_percentage	78
saif_layerStuff{0}.saif_trees{1}.saif_treeCode	SPRUCE
saif_layerStuff{0}.saif_trees{1}.saif_percentage	22
saif_layerStuff{0}.saif_history{0}.saif_actYear1	1990
saif_layerStuff{0}.saif_history{0}.saif_actYear2	1991
saif_layerStuff{0}.saif_history{0}.saif_activityCode	HARVEST
saif_layerStuff{0}.saif_history{1}.saif_actYear1	1992
saif_layerStuff{0}.saif_history{1}.saif_actYear2	1993
saif_layerStuff{0}.saif_history{1}.saif_activityCode	GROOM
saif_layerStuff{0}.saif_history{2}.saif_actYear1	1993
saif_layerStuff{0}.saif_history{2}.saif_actYear2	1994
saif_layerStuff{0}.saif_history{2}.saif_activityCode	HARVEST
saif_layerStuff{1}.saif_layerNumber	2
saif_layerStuff{1}.saif_rank	0
saif_layerStuff{1}.saif_trees{0}.saif_treeCode	SPRUCE


```

Relate TABLE_DEF COLORS DATABASE                                \
    DATABASE_SERVER_TYPE ORACLE                                \
    DATABASE_USER_NAME    scott                                \
    DATABASE_PASSWORD     tiger                                \
    DATABASE_NAME         ""                                   \
    COLORNUM              NUMBER                               \
    COLORNAME              VARCHAR2(30)                         \

# -----
# 3) RELATION_DEF
#   This tells FME how to connect the FME feature (whose fields
#   are on the right hand side) to a row in the database
#   (whose fields are on the left hand side). For this
#   example, we are doing a 1-1 relate, so we expect that the
#   value in the feature for the attribute 'key' will
#   correspond to exactly one row in the database where the
#   COLORNUM has the same value.

Relate RELATION_DEF key_to_color 1:1                            \
    TABLE      COLORS                                         \
    UNIQUE(COLORNUM)                                           \
    JOIN        COLORNUM    TO key                             \
    TRANSFER COLORNAME     TO feature_color_name               \

# -----
# Now create a couple of features to "relate" with...
# -----

FACTORY_DEF * CreationFactory                                  \
    FACTORY_NAME "Creator"                                     \
    OUTPUT FEATURE_TYPE sample key 1                           \
    OUTPUT FEATURE_TYPE sample key 2                           \

# -----
# And do the @Relate, logging the feature before and after
# -----

FACTORY_DEF * SamplingFactory                                  \
    FACTORY_NAME "Sampler"                                     \
    SAMPLE_RATE 1                                              \
    INPUT FEATURE_TYPE *                                       \
        @Log(Before)                                           \
        @Relate(key_to_color, DestReadSrcWrite)                 \
        @Log(After)

```

```
@RemoveAttributes(fme regexp match[,<regexp>]+)
```

Arguments:

Name	Range	Description	Optional
<attrName>	String	The names of the attributes being deleted.	No
<regexp>	String	A regular expression that is used to match the names of the attributes being deleted.	No

This command is used as an attribute filter. All attributes specified in the attribute list are removed from the feature. This can be used to reduce the amount of information stored within a feature. It is also useful for removing undesired components from structures.

If the attribute name is a list attribute ending with {}, then all the attributes in this list are removed.

If the first argument is `fme_regexp_match`, then all following parameters are interpreted as regular expressions (REs) and all attributes that match any of the REs will be removed.

The following rules determine one-character REs that match a single character:

- Any character that is not a special character (to be defined) matches itself.
- A backslash (\) followed by any special character matches the literal character itself (i.e., this "escapes" the special character).
- The "special characters" are:
+ * ? . [] ^ \$

The period (.) matches any character except the newline (for example, ".umpty" matches either "Humpty" or "Dumpty."

- A set of characters enclosed in brackets ([]) is a one-character RE that matches any of the characters in that set. For example, "[akm]" matches either an "a", "k", or "m". A range of characters can be indicated with a dash. For example, "[a-z]" matches any lowercase letter. However, if the

first character of the set is the caret (^), then the RE matches any character *except* those in the set. It does not match the empty string. For example, [^akm] matches any character except "a", "k", or "m". The caret loses its special meaning if it is not the first character of the set.

The following rules can be used to build a multi-character RE:

- A one-character RE followed by an asterisk (*) matches zero or more occurrences of the RE. Hence, [a-z]* matches zero or more lowercase characters.
- A one-character RE followed by a plus (+) matches one or more occurrences of the RE. Hence, [a-z]+ matches one or more lowercase characters.
- A question mark (?) is an optional element. The preceding RE can occur zero or once in the string – no more. For example, xy?z matches either xyz or xz.
- The concatenation of REs is an RE that matches the corresponding concatenation of strings. For example, [A-Z][a-z]* matches any capitalized word.

Finally, the entire regular expression can be anchored to match only the beginning or end of a line:

- If the caret (^) is at the beginning of the RE, then the matched string must be at the beginning of a line.
- If the dollar sign (\$) is at the end of the RE, then the matched string must be at the end of the line.

Inverse Operation

This function has no inverse.

Example

In the example below, the attributes numLanes and highwayNum are removed from all road features.

```
FACTORY_DEF SHAPE SamplingFactory \
  INPUT FEATURE_TYPE road \
    @RemoveAttributes(numLanes, highwayNum) \
  SAMPLE_RATE 1
```

In the example below, any attributes that begin with fme_ are removed from all road features.

```
FACTORY_DEF SHAPE SamplingFactory \
  INPUT FEATURE_TYPE road \
    @RemoveAttributes(fme_regexp_match, ^fme_) \
  SAMPLE_RATE 1
```

@RemoveGeometry

@RemoveGeometry()

Function Type: Feature

Arguments: None

Configuration

This function does not accept configuration lines.

Description

This command is used to purge the geometry of the feature upon which it is called. After this command is executed, the feature contains no coordinates and has its geometry type set to `fme_undefined`.

Inverse Operation

The command has no effect in the inverse (on source side of correlation) direction.

Example

The following example shows how to use the @RemoveGeometry function. It uses the TeeFactory to filter the features that are having their geometry stripped from them. Features have their geometry removed if they have an attribute named `keepGeometry` which has a value of `false`. Features that don't have an attribute named `keepGeometry` or that have a value other than `false` are left untouched.

```
FACTORY_DEF * TeeFactory \
  INPUT FEATURE_TYPE * keepGeometry false \
  OUTPUT FEATURE_TYPE * @RemoveGeometry()
```

@RenameAttributes

```
@RenameAttributes(<change_attr_opt>,<regexp>,<caseType>)
```

```
@RenameAttributes([<targetAttributeName> <sourceAttributeName>]+)
```

```
@RenameAttributes(LIST_ATTR, "value", <attr_list>,<caseType>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<change_attr_opt>	fine_name fine_value fine_name_val	The part of the attribute to be changed. Choices are Name, Value or Both.	No
<regexp>	String	Either the name of the source attribute to change or a regular expression that is used to parse through the set of attribute names and select the ones that need changing.	No
<caseType>	fine_uppercase fine_lowercase fine_titlecase fine_fulltitlecase	The case type to change the selected attribute names and/or values to. Choices are UPPER CASE, lowercase, Title case, and Full Title Case.	No
<targetAttributeName>	String	Target attribute name after change	No
<sourceAttributeName>	String	Source attribute name prior to change	No
<attr_list>	String	Space delimited list of attribute name(s) to change	No
LIST_ATTR	String	Transformer keyword indicating that the next argument is <attr_list> and not <regexp>	No

Configuration

The @RenameAttributes function does not accept configuration lines.

Description

There are three way to use this function:

1. Specify the part of the attribute (name, value or both) that is to be changed, the regular expression to be used to select those attributes to change, and the case type to change to. The regular expression will parse based on attribute names only.
2. Specify the target and the source attribute names
3. Using the CaseChanger transformer, select those attributes to be changed from the list, and what case to change them to.

If there is no specific encoding type associated with the attribute, then it is assumed to be in system encoding.

Inverse Operation

This function does nothing when invoked in the reverse direction, which happens when it appears on the source portion of a transfer specification.

Example

In this example, @RenameAttributes is used to change all attribute values to uppercase.

```
FACTORY_DEF * TeeFactory \
FACTORY_NAME all_toUpper \
INPUT FEATURE_TYPE AttributeCreator_5_OUTPUT_0 \
OUTPUT FEATURE_TYPE all_toUpper_OUTPUT \
@RenameAttributes(value,.*,upper)

@RenameAttributes(LIST_ATTR, "value", "displayType
dockType",upper)

@RenameAttributes("DISPLAYTYPE", "displayType")
```

@ReplaceCoordinates

```
@ReplaceCoordinates(<coordinate>,<searchValue>,<replaceValue>
                    [,<coordinate>,<searchValue>,<replaceValue>]
                    [,<coordinate>,<searchValue>,<replaceValue>])
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<coordinate>	“x”, “y”, or “z”	This is the coordinate that should be scanned for a specific value.	N
<searchValue>	any numeric value or the string “NaN”	This is the exact value that will be searched.	N
<replaceValue>	any numeric value or the string “NaN”	This is the exact value that will replace any search values that are found.	N

Configuration

The @ReplaceCoordinates function does not accept configuration lines.

Description

This function scans the geometry of the feature and replaces specific values with new values. The value NaN is represented by the special token NaN.

Inverse Operation

The inverse of this function searches for the replaceValue and replaces it with the searchValue.

Example

The example below replaces every Z value of "not a number" (unknown) with "0":

```
@ReplaceCoordinates(z,NaN,0)
```

The example replaces every X value of "-999" with "not a number" (unknown):

```
@ReplaceCoordinates(x,-999,NaN)
```


The example below replaces every X value of 0 with 100, every Y value of 0 with 500, and every Z value of -1 with 0:

```
@ReplaceCoordinates(x,0,100,y,0,500,z,-1,0)
```

@Reproject

```
@Reproject (<sourceCS>, <destCS>[, COORDINATES_ONLY] [, <rasterInterpolationType>] [, <rasterCellSize>])
```

or

```
@Reproject (<reprojection-Engine>, <sourceCS>, <destCS>[, <param1>[, <param2>...]], --[, COORDINATES_ONLY] [, <rasterInterpolationType>] [, <rasterCellSize>])
```

or

```
@Reproject (<sourceCS>, <destCS>, <xFieldName>, <yFieldName>)
```

or

```
@Reproject (<reprojection-Engine>, <sourceCS>, <destCS>[, <param1>[, <param2>...]], --, <xFieldName>, <yFieldName>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<sourceCS>	Coordinate System Name	The name of the source coordinate system.	No
<destCS>	Coordinate System Name	The name of the destination coordinate system.	No
<reprojectionEngine>	FME, ESRI, GTRANS	The reprojection engine to use when reprojecting coordinates.	Yes
<param1>	Varies based on reprojection engine	For the ESRI reprojection engine, this is the optional geotransformation name. For the GTRANS reprojection engine, this is the translation file path.	Yes

Name	Range	Description	Optional
<param2>	Varies based on re-projection engine	For the ESRI reprojection engine, this is the geotransformation direction (either “forward” or “reverse”). For the GTRANS reprojection engine, this is the inverse translation file path.	Yes
<rasterCellSize>	SquareCells, StretchCells, PreserveCells	For raster reprojection, determines how the cell size of the output raster will be calculated. See the Reprojector transformer help text.	Yes
<rasterInterpolationType>	NEARESTNEIGHBOR, BILINEAR, BICUBIC, AVERAGE4, AVERAGE16	For raster reprojection, determines how each cell is interpolated. See the Reprojector transformer help text.	Yes
<xFieldName>	Attribute Name	The name of the attribute containing the x coordinate value.	Yes
<yFieldName>	Attribute Name	The name of the attribute containing the y coordinate value.	Yes

Description

The @Reproject function is only required in rare cases during FME translations. The recommended approach to performing coordinate conversions is to use the <readerKeyword>_COORDINATE_SYSTEM and <writerKeyword>_COORDINATE_SYSTEM. Search *Coordinate System Support* in the *FME Universal Translator on-line help*.

This function provides direct access to the FME’s coordinate conversion capabilities. With this function, feature coordinates may be converted from one coordinate system to another during data translation.

When executed in the forward direction on the destination line of a feature correlation or on an INPUT or OUTPUT clause of a factory definition, the coordinate conversion operation is performed in the forward direction. In this case, coordinates are converted from the source coordinate system to the destination coordinate system.

If xFieldName and yFieldName are specified, then a single point coordinate conversion will be performed on the values of these attributes. The resulting

new coordinates will be stored back into these same attributes. In this case, the feature's geometry coordinates are left untouched.

Tip

The field name parameters are usually used to convert the coordinates of an inside point, often stored as an attribute of a polygonal feature.

When `xFieldName` and `yFieldName` are not specified, the coordinate conversions are performed on the geometry of the feature. This is the usual mode of operation. In this mode, if the input features have a coordinate system defined, that system is used as the source of the reprojection, and the `sourceCS` parameter is ignored (`destCS` in the case of inverse operation).

Reprojection of arcs, ellipses, and text is handled differently based on whether the input features are represented with classic or enhanced geometry. For classic geometry, FME performs additional operations just prior to reprojecting the feature. These operations typically include stroking arcs and ellipses (converting them to lines and polygons), and may be skipped by specifying the `COORDINATES_ONLY` flag. With the `COORDINATES_ONLY` flag, only the center points of classic arcs and ellipses are reprojected. For enhanced geometry, the additional operations and `COORDINATES_ONLY` flag do not apply. All of the relevant control points and properties of arcs, ellipses, and text are reprojected. Arcs and ellipses are only stroked when maintaining them as arcs and ellipses would introduce significant distortion.

`interpolationType` only has an effect on raster.

`NearestNeighbor` is the fastest but produces the poorest image quality. `Bicubic` is the slowest but produces the best image quality. `Bilinear` provides a reasonable intermediate option. `Average4` and `Average16` have a performance similar to `Average4` and can be useful for numeric rasters such as DEMs.

`rasterCellSize` also only applies to raster features. If it is set to `StretchCells`, the cell size of the raster will be adjusted to maintain the same number of rows and columns in the reprojected raster as there were in the input raster. If it is set to `SquareCells`, the number of rows and columns as well as the spacing will be changed to maintain approximately the same cell ground area and form square cells where the horizontal and vertical cell sizes are equal. Like the `SquareCells` option, `PreserveCells` will change both the number of rows and columns and the spacing to maintain cell ground area, but will also try to preserve the original cell aspect ratio, taking into account any warping caused by the reprojection.

Note: If the source coordinate system is not fixed and may change from feature to feature, and the features themselves have been tagged with a coordinate system from the reader that produced them, then a single Reprojector may still be able to be used. In such a case, both the source and destination coordinate system can be set to the same value – the

destination coordinate system – and the desired behavior will be accomplished.

This function is currently unaffected by raster and/or palette sub-selection.

Inverse Operation

When executed in the reverse direction source line of a feature correlation, the coordinate conversion operation converts coordinates from the destination coordinate system to the source coordinate system.

Example

In the example below, the correlation line is used to convert the Design features from UTM zone 10 based on NAD27 datum to lat/long coordinates based on NAD83 datum, for only the road features in the data set.

```
# Specify the correlation lines from IGDS to SHAPE.
# The Reproject command is placed on the SHAPE line.
# When converting data from IGDS to SHAPE the
# coordinates are converted from UTM zone 10 based on
# NAD 27 to Lat/Long coordinates based on NAD 83. When
# converting data from SHAPE to IGDS the coordinates
# are converted from Lat/Long coordinates (NAD83) to
# UTM zone 10 (NAD 27).
```

```
IGDS 23 igds_type igds_line
SHAPE roads @Reproject(UTM27-10, LL83)
```

The second example illustrates how to use the coordinate conversions to change coordinate values stored as attributes of a feature. When translating from SAIF to Shape, the coordinate geometry is translated as in the previous example. The features also have inside point coordinates stored in the attributes `insideX` and `insideY` which must also be translated. These are done using a second `@Reproject` command invocation.

```
SAIF Lake::MOF insideX %x insideY %y
SHAPE lake insideX %x insideY %y \
  @Reproject(UTM27-10, LL83) \
  @Reproject(UTM27-10, LL83, insideX, insideY)
```

The third example takes a raster image in UTM84-17N and reprojects it to LL. The `interpolationType` is set to `Bicubic` to get the most accurate result, and the `rasterCellSize` is set to `StretchCells` to get an output raster with the same number of rows and columns as the input raster.

```
GEOTIFF utm @Reproject(UTM84-17N, LL, NearestNeighbor,
StretchCells)
```

@ReprojectAngle

```
@ReprojectAngle(<angle in source CS>,           \  
                <distance in source CS>         \  
                <sourceCS>, <destCS>)
```

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<angle in source CS>	Float	Angle of distance in source coordinate system. The angle is measured counter clockwise from horizontal and is specified in degrees.	No
<distance in source CS>	Float	Distance specified in source coordinate system coordinates.	No
<sourceCS>	Coordinate System Name	The name of the source coordinate system.	No
<destCS>	Coordinate System Name	The name of the destination coordinate system.	No

Description

This function is used to convert an angle from one coordinate system to another. The function uses the first coordinate of the passed-in feature and returns the angle of the specified line at the determined angle in the destination coordinate system. This is useful for converting data such as text heights and widths that are established in many systems in the destination coordinate system units.

This function assumes all coordinate systems are Cartesian.

Inverse Operation

When executed in the reverse direction on the source line of a feature correlation, the angle conversion operation is performed in the reverse direction.

Example

In the example below, all `mif_text` features have an attribute `new_angle` added to them. The call to the function specifies the source coordinate system as UTM zone 10 based on the NAD27 datum and the destination coordinate system as lat/long coordinates based on the NAD83 datum.

```

FACTORY_DEF MIF SamplingFactory \
  INPUT FEATURE_TYPE mif_text \
    new_angle @ReprojectAngle(&mif_rotation, \
                              &mif_text_width, \
                              UTM27-10, LL83) \
  SAMPLE_RATE 1

```

Note When features are translated from one coordinate system to another within the FME, it is not necessary to reproject angles in this way. Coordinates, as well as special attributes such as text height, rotation, etc., are automatically reprojected.

@ReprojectLength

```
@ReprojectLength(<distance in source CS>           \  
                  <angle in source CS>,           \  
                  <sourceCS>, <destCS>)
```

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<distance in source CS>	Float	Distance specified in source coordinate system.	No
<angle in source CS>	Float	Angle of distance in source coordinate system. The angle is measured counter-clockwise from horizontal and is specified in degrees.	No
<sourceCS>	Coordinate System Name	The name of the source coordinate system.	No
<destCS>	Coordinate System Name	The name of the destination coordinate system.	No

Description

This function is used to convert a distance in one coordinate system to a distance in another. The function uses the first coordinate of the passed-in feature and returns the length of the line at the specified angle in the destination coordinate system. This is useful for converting data such as text heights and widths that are measured in many systems in the destination coordinate system units.

This function assumes all coordinate systems are cartesian.

Inverse Operation

When executed in the reverse direction on the source line of a feature correlation, the length conversion operation is performed in the reverse direction.

Example

In the example below, all `mif_text` features have an attribute `new_width` added to them. The call to the function specifies the source coordinate system as UTM zone 10 based on the NAD27 datum and the destination coordinate system as lat/long coordinates based on the NAD83 datum.

```

FACTORY_DEF MIF SamplingFactory                                \
  INPUT FEATURE_TYPE mif_text                                  \
    new_width @ReprojectLength  (&mif_text_width,            \
                                  &mif_rotation,              \
                                  UTM27-10, LL83)              \

```


@ResampleRaster

```
@ResampleRaster(DIMENSIONS, <rows>, <columns> \
    [, <interpolation type> ])\n@ResampleRaster(CELL_SIZE, <x cell size>, <y cell size> \
    [, <interpolation type> ])
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<data type>	DIMENSIONS CELL_SIZE	The first argument specifies whether the next two arguments indicate the desired dimensions or cell size for the raster.	No
<rows>	Integer	The number of rows that the raster should have after resampling.	No (first form only)
<columns>	Integer	The number of columns that the raster should have after resampling.	No (first form only)
<x cell size>	Real number	The horizontal size of the raster's cells, after resampling.	No (second form only)
<y cell size>	Real number	The vertical size of the raster's cells, after resampling.	No (second form only)
<interpolation type>	NEARESTNEIGHBOR BILINEAR BICUBIC AVERAGE4 AVERAGE16	The type of interpolation to use. Default is NEARESTNEIGHBOR.	Yes

Configuration

This function does not accept configuration lines.

Description

The `@ResampleRaster` function is used to resample a raster, given either the desired dimensions of the raster (in rows and columns) or the desired cell size. Cell values are interpolated in order to change the raster to the specified (larger or smaller) size. The resampling is done using a nearest-neighbor interpolation by default. Other interpolation types can be specified using the optional argument.

This function is currently unaffected by raster and/or palette sub-selection.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a use of the `@ResampleRaster` function. In this example, the `SamplingFactory` is used to filter the incoming features, while the `@ResampleRaster` function resamples the rasters to a new size of 500 rows by 600 columns.

```
FACTORY_DEF * SamplingFactory           \
    SAMPLE_RATE 1                       \
    INPUT FEATURE_TYPE *                \
        @ResampleRaster(DIMENSIONS, 500, 600) \
    OUTPUT FEATURE_TYPE *
```

The next example is identical, except that it uses a bilinear interpolation:

```
FACTORY_DEF * SamplingFactory           \
    SAMPLE_RATE 1                       \
    INPUT FEATURE_TYPE *                \
        @ResampleRaster(DIMENSIONS, 500, 600, BILINEAR) \
    OUTPUT FEATURE_TYPE *
```

This example does resampling by specifying cell sizes instead of rows and columns. As in the above example, the raster is aligned to (10, 20):

```
FACTORY_DEF * SamplingFactory           \
    SAMPLE_RATE 1                       \
    INPUT FEATURE_TYPE *                \
        @ResampleRaster(CELL_SIZE, 0.07, 0.08) \
    OUTPUT FEATURE_TYPE *
```

@ResolveRasterPalettes

Note: This function has been renamed from @RasterPaletteResolver.

@ResolveRasterPalettes()

Function Type: Feature

Arguments: None

Configuration

This function does not accept configuration lines.

Description

The @ResolveRasterPalettes function is used to ensure that the source band type is continuous. If the source is classified, it will be resolved to be continuous without changing the original data source. If the source is continuous, nothing will be done and the data will be passed straight through.

This function supports sub-selection of raster bands and palettes.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

The following example shows a use of the @ResolveRasterPalettes function. In this example, the SamplingFactory is used to pass through every feature and those that have a classified band type will be resolved by the @ResolveRasterPalettes function to be continuous. The features that have a continuous band type will not be affected.

```

FACTORY_DEF * SamplingFactory                                \
    SAMPLE_RATE 1                                           \
    INPUT FEATURE_TYPE *                                    \
        @ResolveRasterPalettes()                            \
    OUTPUT FEATURE_TYPE *
```

@Rotate2D

```
@Rotate2D(<degrees> [, <rotateX>, <rotateY>])
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<degrees>	Real Value	Specifies the angle that the feature will be rotated, measured in degrees counterclockwise.	No
<rotateX>	Real Value	The x-coordinate about which features are rotated. If not specified, the feature will be rotated about the origin. This parameter does not apply when rotating raster geometries.	Yes
<rotateY>	Real Value	The y-coordinate about which features are rotated. If not specified, the feature will be rotated about the origin. This parameter does not apply when rotating raster geometries.	Yes

Configuration

This function does not accept configuration lines.

Description

This command rotates features in a counterclockwise direction about the specified point by the determined angle's number of degrees. If the point around which the rotation is to be performed is not specified, then the feature will be rotated about the origin.

Feature with raster geometries support the angle of rotation but ignore the rotateX and rotateY parameters since raster rotation is around the raster origin which is the upper left corner of the raster.

Inverse Operation

The function rotates the feature in the clockwise direction when invoked on the source line of a transformation specification.

Example

In the example below, the building is rotated 90 degrees about the point identified by the attributes `centroidX`, and `centroidY`. When going from Shape to MIF, the rotation is in the counterclockwise direction and when going from MIF to Shape, the rotation is in the clockwise direction.

```
SHAPE building
MIF building @Rotate2D(90, &centroidX, &centroidY)
```

@RoundOffCoords

@RoundOffCoords(<axis>, <precision>)

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<axis>	(x y z xyz)	The first parameter controls the axis of the coordinates being rounded. If xyz is specified, then the respective precision values will be used for each axis.	No
<precision>	Real Number	<p>The second parameter controls the number of decimal places of precision to which the coordinates are rounded. A value of 0 causes all coordinates along the specified axis to be rounded to the nearest integer. If the axis is specified as xyz, then this precision value will be used for the x axis. A value of 1 causes rounding to the nearest tenth of a unit. Negative values are allowed. A value of -1 causes rounding to the nearest 10. The precision can be a floating point number. This can be used to snap all coordinates to an arbitrary grid. The formula used to do the rounding is:</p> <pre>factor = pow(10.0, <precision>) newCoordValue = factor * (RoundToInt(oldCoordValue/factor))</pre> <p>So if precision is set to the $\log_{10}(2) = 0.3010299956639812$, all coordinates are rounded to the nearest 1/2 unit. Similarly, if precision is set to $\log_{10}(0.5) = 0.3010299956639812$, all coordinates are rounded to the nearest even unit.</p>	No
<y_precision>	Real Number	Similar to the precision parameter, this controls the number of decimal places of precision to which the coordinates are rounded for the y axis. This is only applicable if the axis parameter is set to xyz.	Yes
<z_precision>	Real Number	Similar to the precision parameter, this controls the number of decimal places of precision to which the coordinates are rounded for the z axis. This is only applicable if the axis parameter is set to xyz.	Yes

Configuration

This function does not accept configuration lines.

Description

This function rounds off the coordinates of the passed axis to the passed-in precision. It may be used to remove superfluous decimal points in the coordinates when they are destined for an ASCII output file.

Any consecutive points that become duplicates as a result of the rounding are thinned by removing the redundant points.

If the `FME_GEOMETRY_HANDLING` directive is set to “yes” in the mapping file, arcs have their start and end points rounded off, and ellipses aren't touched; otherwise, arcs and ellipses have their center points rounded off.

Inverse Operation

The inverse operation is the same as the forward operation.

Example

In the example below, the x and y axes have their coordinates rounded off to the nearest hundredth of a ground unit after being projected from lat/long to UTM. The z axis is not touched.

```

FACTORY_DEF SAIF SamplingFactory          \
SAMPLE_RATE 1                             \
INPUT FEATURE_TYPE *                       \
    @Reproject(LL-83,UTM10-83)             \
    @RoundOffCoords(x,2)                   \
    @RoundOffCoords(y,2)                   \

```

In this second example, the x coordinates are rounded to the nearest half unit, and the y coordinates are rounded to the nearest multiple of 5. The z coordinates are truncated to the nearest whole unit.

```

MACRO Log10_2      0.3010299956639812
MACRO MinusLog10_5 -0.69897000433601886
FACTORY_DEF SAIF SamplingFactory          \
SAMPLE_RATE 1                             \
INPUT FEATURE_TYPE *                       \
    @RoundOffCoords(x, $(Log10_2))         \
    @RoundOffCoords(y, $(MinusLog10_5))    \
    @RoundOffCoords(z, 0)                  \

```

@Scale

```
@Scale(<factor>[, TEXT_LOCATION_ONLY])
@Scale(<xFactor>, <yFactor>[, TEXT_LOCATION_ONLY])
@Scale(<xFactor>, <yFactor>, <zFactor>[, TEXT_LOCATION_ONLY])
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<factor>	Real Value	All x, y, and z coordinates are multiplied by these scaling factors.	No
<xFactor>	Real Value	Specifies the scale factor for the x component of the feature coordinates.	No
<yFactor>	Real Value	Specifies the scale factor for the y component of the feature coordinates.	No
<zFactor>	Real Value	Specifies the scale factor for the z component of the feature coordinates.	No
TEXT_LOCATION_ONLY	N/A	If this optional parameter is present, only the location of text will be scaled; otherwise, both the location and text size will be scaled.	Yes

Configuration

This function does not accept configuration lines.

Description

This command scales the coordinates of the passed-in feature by the multipliers specified. If just one value is given, then x, y, and z will be scaled by that amount. If two values are given, then x and y will be scaled as specified, and the z component will be left untouched. If three values are given, then x, y, and z will be scaled. If the optional `TEXT_LOCATION_ONLY` flag is specified, then only the text location (not the text size) will be scaled. If this flag is not specified, then both the size and location of text will be scaled. If the feature is an arc or ellipse, then the values for FME-specific attributes `fme_primary_axis` and `fme_secondary_axis` will also be scaled accordingly.

This command also supports raster features and supports sub-selection of raster bands and palettes. Since raster features in FME are always north-facing, the `<xFactor>` and `<yFactor>` values should only be positive numbers. If they are

negative, this command will adjust them to their absolute values. The <xFactor>, <yFactor> and <zFactor> values must not be zero.

Inverse Operation

The function divides the features by the specified scaling factor(s).

Example

In the example below the building coordinates are multiplied by a factor of 2 when going from Shape to MIF. When going from MIF to Shape, the building coordinates are divided by a factor of 2.

```
SHAPE building
MIF building @Scale(2.0)
```

@SDEsql

Note: This function is not supported by FME Base Edition.

```
@SDEsql (<streamName>, (<rdbms-statement> |
FME_FREE_STREAM) )
@SDEsql (FME_FREE_ALL_STREAMS)
```

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<streamName>	String	The name of a stream defined on a configuration line. This stream is where the SQL statement is executed.	No
<rdbms-statement>	String FME_FREE_STREAM	The RDBMS statement that is to be executed on the defined stream. If FME_FREE_STREAM is specified instead of an RDBMS statement, then the stream specified in the first parameter will be deleted.	No

Configuration

```
SDEsql<streamName> <host> <instance> <database> <userID>\
<password>
```

Name	Range	Description	Optional
<streamName>	String	The name of the SDE stream upon which the SQL statement will be executed.	No

Connection Parameters

There are two possible ways to connect to SDE: connecting to the ArcSDE service, which in turn communicates with the underlying database (a three-tier architecture), and connecting to the underlying database directly (a two-tier architecture). With a direct connection (the two-tier architecture), ArcSDE does not need to be installed and an ArcSdeServer license is only needed if writing to the database; reading does not require a license.

Regular Connection

The connection parameters needed for a regular connection are as follows:

Name	Range	Description	Optional
<host>	String	The name of the host computer upon which the commands are to be executed.	No
<instance>	String	The name of the SDE instance to use.	No
<database>	String	The name of the SDE database to use. If unused by the underlying RDBMS, then this value is ignored.	No
<userID>	String	The user ID used to access the database.	No
<password>	String	The password for the user ID.	No

Direct Connection

The parameters needed to make a direct connection to SDE depend on the underlying database. In order to make a direct connection, the SDE must be of the same major version as the client libraries with which the @SDEsql was built. For example, a direct connection to an ArcSDE 9.1 instance could be made with an @SDEsql built using 9.0 libraries, but a connection could not be made to an ArcSDE 8.3 instance using the same @SDEsql.

Underlying Database	Name	Range	Description
Oracle (option 1)	<database>	String	Any value can be specified as the value does not get used; however, a value must be supplied.
	<instance>	String	sde:oracle or sde:oracle9i (for 9i connections to use the right driver)
	<userID>	String	<username>
	<password>	String	<password>@<Oracle Net Service Name>
Oracle (option 2)	<database>	String	Any value can be specified as the value does not get used; however, a value must be supplied.
	<instance>	String	sde:oracle: ;local=<sqlnetalias>
	<userID>	String	<username>
	<password>	String	<password>

Underlying Database	Name	Range	Description
MS SQL Server	<database>	String	<database_name>
	<instance>	String	sde:sqlserver:<SQL Server Instance Name> or sde:sqlserver:<SQL Server Instance Name>\<Named Instance> (for connecting to a named instance)
	<userID>	String	<username>
	<password>	String	<password>
DB2 (option 1)	<database>	String	<db alias name specified through DB2 Configuration Assistant>
	<host>	String	remote (if client application is remote, otherwise do not specify)
	<instance>	String	sde:db2
	<userID>	String	<username>
	<password>	String	<password>
DB2 (option 2)	<database>	String	Any value can be specified as the value does not get used; however, a value must be supplied.
	<host>	String	remote (if client application is remote, otherwise do not specify)
	<instance>	String	sde:db2:<db alias name specified through DB2 Configuration Assistant>
	<userID>	String	<username>
	<password>	String	<password>
Informix	<database>	String	Any value can be specified as the value does not get used; however, a value must be supplied.
	<host>	String	remote (if client application is remote, otherwise do not specify)
	<instance>	String	sde:informix:<odbc data source name>
	<userID>	String	<username>
	<password>	String	<password>

Description

This function enables the FME to execute arbitrary database commands through an SDE connection to the database. The SDE passes the SQL statement to the underlying RDBMS for execution. The purpose of the command is to provide access to SQL statements which enable row updates or other such statements. If you want to search a database and perform operations on the feature retrieved, then you should consult both the *ArcSDE Reader/Writer* chapter in the *FME Readers and Writers* manual, and the *SDE30QueryFactory* section in this manual. This command has no return value and is expected to be used as a means of performing arbitrary SQL statements.

Once you have finished using a stream, you may want to free it before the end of the translation. In fact, this may be necessary if the maximum number of concurrent streams allowed by SDE has been reached and there is a need to create more streams. To delete a particular stream, use the first form of the function and specify `FME_FREE_STREAM` as the second parameter. To delete all streams, use the second form of the function.

Assumptions

This function only works with SDE 3.x/ArcSDE 8.x/9.0. There is no support for this command on versioned tables or layers. Using this command on a versioned table or layer will result in the SQL statement being applied only to the base table. No changes will be made to the additions table or to the deletions table, and no database states will be created.

Inverse Operation

The command performs the same operation in the inverse direction.

Example

The following example uses the `@SDEsql` function to update a date field in an Oracle table named `OPER_STATUS`. The example shows how macro values can be used to control the connection characteristics to the SDE for the command. The command specified is based on Oracle and, as such, may not run on other RDBMSes. The command is highlighted in bold. While it is split over many lines below, it must be on the same line in an FME mapping file. Notice how the value for database is specified even though the underlying RDBMS doesn't use it.

```
SDEsql gridUpdateStream $( _SDE3Server)           \
                        $( _SDE3Instance)          \
                        not_used                     \
                        $( _SDE3User)               \
                        $( _SDE3Password)
```

```
FACTORY_DEF * SamplingFactory \
FACTORY_NAME UPDATE_GRID \
INPUT FEATURE_TYPE * tag gridRecords \
@SDEsql(gridUpdateStream,"update OPER_STATUS SET
DATE_MODIFIED=TO_DATE('$(RUN_DATE_TIME)',
'YYYYMMDDHH24MISS') WHERE ID='&ID'") \
SAMPLE_RATE 0
```


@SearchList

```
@SearchList(<listName>, <value>, [FIRST_MATCH|
FIRST_NOT_MATCHED|REGULAR_EXPRESSION|
REGULAR_EXPRESSION_ENCODED|
FIRST_NOT_MATCHED_ENCODED|FIRST_MATCHED_ENCODED|
FIRST_LESS_THAN_ENCODED|FIRST_GREATER_THAN_ENCODED|
FIRST_LESS_THAN_OR_EQUAL_ENCODED|
FIRST_GREATER_THAN_OR_EQUAL_ENCODED])
```

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<listName>	Any Feature Attribute List	The function searches this attribute list for the <value>. The list name should contain a { } pair to indicate it is a list.	No
<value>	String or Number	The function searches the attribute list for this value.	No
search type	FIRST_MATCH FIRST_NOT_MATCHED REGULAR_EXPRESSION	If there is no value specified or it is FIRST_MATCH, then the first matching list entry’s index is returned. If it is FIRST_NOT_MATCHED, then the first non-matching list entry’s index is returned. The search type REGULAR_EXPRESSION is like FIRST_MATCH but it uses the regular expression to find a match. Note: Some special characters such as “@” and “&” have special meanings in FME and may not produce expected results if used in the regular expression string.	Yes
search type (ENCODED options)	REGULAR_EXPRESSION_ENCODED FIRST_NOT_MATCHED_ENCODED FIRST_MATCHED_ENCODED FIRST_LESS_THAN_ENCODED FIRST_GREATER_THAN_ENCODED FIRST_LESS_THAN_OR_EQUAL_ENCODED FIRST_GREATER_THAN_OR_EQUAL_ENCODED	Expands on the search types above to allow special characters to be used in the search string. For more information, search <i>Substituting Strings in Mapping Files</i> in the FME Fundamentals manual online help file.	Yes

Configuration

This function does not accept configuration lines.

Description

This function searches the attribute list given as its first parameter for the value passed in as the second parameter. It will return the index of the first matching list entry or -1 if no match is found.

For example, if this feature enters this function:

```
somelist{0}.length = 7.3
somelist{0}.kind = 'paved'
somelist{1}.length = 8.4
somelist{1}.kind = 'smooth'
somelist{1}.lanes = 2
somelist{2}.length = 1.1
somelist{2}.kind = 'rough'
```

and the `somelist{}.kind` list attribute is searched for the value 'smooth' with the Search Type set to `FIRST_MATCH`, then the index attribute would be set to 1.

If search type is `FIRST_MATCH`, it uses a regular expression to search for a first matching entry in the list.

If search type is `FIRST_NOT_MATCHED`, it will return the index of the first non-matched entry in the list.

Advanced Regular Expressions (AREs) are supported. Search the *FME Functions and Factories* on-line help for a complete description of AREs. In brief, an ARE is one or more branches, separated by '|', matching anything that matches any of the branches.

`FIRST_LESS_THAN_ENCODED`, `FIRST_GREATER_THAN_ENCODED`, `FIRST_LESS_THAN_OR_EQUAL_ENCODED`, `FIRST_GREATER_THAN_OR_EQUAL_ENCODED` similarly search for the first element in the list that satisfies the criteria. Numerical comparisons are used if both the list element and the search value can be converted to floating point numbers; otherwise, string comparisons are used. When any of the `ENCODED` options are used, additional special characters are supported. For more information, search *Substituting Strings in Mapping Files* in the FME Fundamentals manual online help file.

Inverse Operation

This function has no inverse.

Example

Example

Example

Example

@SecondOrderConformal

@SecondOrderConformal (<A>, , <C>, <D>, <E>, <F>, <G>, <H>) for 2D and

@SecondOrderConformal (<A>, , <C>, <D>, <E>, <F>, <G>, <H>, <I>) for 3D

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<A>, , <C>, <D>, <E>, <F>, <G>, <H>	Real number	Coefficients of the 2D conformal transformation. The transformation results in the x and y coordinates being modified by: $x' = C + E(x-A) - F(y-B) + G((y-B)^2 - (x-A)^2) - 2H(x-A)(y-B)$ $y' = D + E(y-B) + F(x-A) + H((y-B)^2 - (x-A)^2) + 2G(x-A)(y-B)$	No
<A>, , <C>, <D>, <E>, <F>, <G>, <H>, <I>, <J>, <K>, <L>	Real number	Coefficients of the 3D conformal transformation. The transformation results in the x and y coordinates being modified identical to above, and in addition, the coordinates being modified by: $z' = z + I$	No

Configuration

The @SecondOrderConformal function does not accept configuration lines.

Description

This function performs a second-order conformal transformation to the feature coordinates upon which it is invoked. Depending on the input geometry and input parameters, a 2D or 3D transformation is performed.

This transformation is non-linear, and thus does not preserve the shape of geometries. However, we preserve the integrity of the input geometry (i.e. input lines will be kept as lines with modified coordinates with respect to the transformation).

If the parameters G and H are zero, then this function performs an equivalent affine (linear) transformation. Using `@Affine` in this case will achieve the same result.

Note This operation is not supported for 3D features.

Inverse Operation

The inverse is not supported by this function.

Example

In the example below, a 2D transformation is applied to all features as they flow through the FME:

```
FACTORY_DEF DWG SamplingFactory SAMPLE_RATE 1 \
INPUT FEATURE_TYPE * \
    @SecondOrderConformal(0,0,0,0,1.1,1.1,0.01,0.01)
```


@SelectRaster

```
@SelectRaster(<codestring>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<code><codestring></code>	String	A parseable string that specifies which bands and palettes that wish to be selected. The format of the string is B P, where B is the band number and P is the palette number of the band and palette to be operated on. B and P are separated by a space. Multiple palettes for a band can be specified delimited by commas. Multiple band-palette pairs can be specified delimited by semicolons. The directive ALL can be used in place of band and palette numbers to select all bands or all palettes on a certain band.	No

Configuration

This function does not accept configuration lines.

Description

This function is used to select specific bands and palettes of a raster for subsequent function and factory operations. The bands and palettes are selected using the band and palette indices, specified in a string.

The format of the string is B P (separated by a space), where B is the band number and P is the palette number of the band and palette to be operated on. The function will only accept alphanumeric characters and valid symbols in the code string. The code string accepts commas (,), spaces(), and semicolons (;).

Multiple palettes for a band can be specified as delimited by commas. Multiple band-palette pairs can be specified as delimited by semicolons. The directive `ALL` can be used in place of band and palette numbers to select all bands or all palettes on a certain band. Specific palettes cannot be selected on `ALL` bands.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example 1

The following example shows a typical use of @SelectRaster. SamplingFactory is used to filter the incoming features, while @SelectRaster selects band 0 with palettes 0 and 2.

```

FACTORY_DEF * SamplingFactory      \
    SAMPLE_RATE 1                  \
    INPUT FEATURE_TYPE *           \
        @SelectRaster(0 0,2)       \
    OUTPUT FEATURE_TYPE *

```

Example 2

In this example, @SelectRaster selects band 0 with all its palettes.

```

FACTORY_DEF * SamplingFactory      \
    SAMPLE_RATE 1                  \
    INPUT FEATURE_TYPE *           \
        @SelectRaster(0 ALL)       \
    OUTPUT FEATURE_TYPE *

```

Example 3

In this example, @SelectRaster selects all bands and palettes.

```

FACTORY_DEF * SamplingFactory      \
    SAMPLE_RATE 1                  \
    INPUT FEATURE_TYPE *           \
        @SelectRaster(ALL)         \
    OUTPUT FEATURE_TYPE *

```

Example 4

The following is a more complex example of using @SelectRaster. Multiple bands and palettes are selected, namely band 0 with palette 1, band 1 with palettes 0 through 2, and band 2 with all of its palettes.

```

FACTORY_DEF * SamplingFactory      \
    SAMPLE_RATE 1                  \
    INPUT FEATURE_TYPE *           \
        @SelectRaster(0 1;1 0,1,2;2 ALL) \
    OUTPUT FEATURE_TYPE *

```


@Snip

```
@Snip(<fromPosition>,<toPosition>[, (DISTANCE|PER-
CENTAGE|VERTEX|<normalizedLength>)] [, (2D|3D) )
```

```
@Snip(<fromXCoord>,<fromYCoord>,<toXCoord>,<toYCo-
ord>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<fromPosition>, <toPosition>	Real number	Define the distances to retain, mea- sured from the first coordinate of the line, of the beginning and end of the portion of the line. Each of these can be a numeric value or one of the liter- al strings "start" or "end", to refer to the first coordinate or last coordinate of the line, respectively. A value of "0" is equivalent to "start", and a val- ue of "-1" is equivalent to "end".	No
<normalizedLength>	Positive real	Specifies the upper bound on the range in which <fromPosition> and <toPosition> are specified. A length specifier of PERCENTAGE is equivalent to specifying a normalized length of 100.0; in this case <toPo- sition> and <fromPosition> specify percentage of distance along the line. A length specifier of VERTEX indicates that the <fromPosition> and <toPosition> are 0-relative indices of the first and last vertices of the line (inclusive) to be contained in the resulting feature's geometry. If a textual keyword or specific <nor- malizedLength> is not specified, the locations are assumed to be spec- ified in ground units.	Yes

Name	Range	Description	Optional
<fromXCoord>, <fromYCoord>	Real number	Specifies an (x,y) coordinate from which to begin snipping. If the coordinate is not directly on the line being snipped, the given (x,y) value will be snapped to the nearest coordinate which is on the line.	No
<toXCoord>, <toYCoord>	Real number	Specifies an (x,y) coordinate from which to begin snipping. If the coordinate is not directly on the line being snipped, the given (x,y) value will be snapped to the nearest coordinate which is on the line.	No

Configuration

This function does not accept configuration lines.

Description

This function is used to shorten a line by chopping off the ends. It only operates on features with simple line geometry and polygons without holes.

Features

The parameters specify a starting and ending point for the snipping; after execution, the feature's geometry will be a line representing the portion of the original line between those two positions, inclusive of the endpoints. New coordinates will be generated at the beginning or end of the line, if there are not already coordinates at exactly the specified positions. If the line contains three-dimensional coordinates, the Z value at each endpoint will be interpolated linearly from the original feature's coordinates between which the endpoint exists.

The start and end of the snipping can be specified in one of two forms. In the first form, the locations are specified as a measured distance from the first coordinate of the original features. By default, the measurement is taken as the same units as the coordinates of the feature. However, a third parameter may be specified to indicate how the distance is to be interpreted. It may be a number or one of the special keywords: `DISTANCE`, `PERCENTAGE`, or `VERTEX`.

If the distance interpretation is specified as a number, it indicates that the <fromPosition> and <toPosition> are expressed with the assumption that the line has length equal to the given <normalizedLength>. In other words, the positions are in the numerical range of [0, <normalizedLength>], where 0 is the first vertex of the original line and <normalizedLength> is its final vertex.

An interpretation of `DISTANCE` is the default if no distance specifier is given. It indicates that `<fromPosition>` and `<toPosition>` are measured in ground units, from the initial vertex of the line.

An interpretation of `PERCENTAGE` indicates that `<fromPosition>` and `<toPosition>` express the distance as a percentage of the total length of the line.

An interpretation of `VERTEX` indicates that `<fromPosition>` and `<toPosition>` specify the indices of actual vertices within the original line feature. The index is 0-relative; that is, the initial vertex of the line is at index 0.

The final optional parameter in the first form of the form of this function tells it how to measure its length. If the input feature's geometry is two-dimensional, or the measurement specifier is given as `2D` (the default), then all measurements are taken in the (x,y) plane. If the function is operating on a three-dimensional line, and the measurement specifier is given as `3D`, the Z coordinate value will be taken into account when computing the length of each line segment.

Note The `2D/3D` specifier does not have any effect on two-dimensional lines, or when `<fromPosition>` and `<toPosition>` are being interpreted using the `VERTEX` option.

The second form by which to specify the snipping bounds is as a pair of (x,y) coordinate values. If the endpoints for the snipping are provided in this form, the nearest location on the feature's line is used as the endpoint of the resulting geometry.

Note `@Snip` currently only operates on plain lines and simple polygons. Attempts to `@Snip` points, donuts, aggregates, arcs, or other geometry will have no effect.

@Split

```
@Split(<splitStr>, (<delimiter>|<formatSpec>), \
    <attrName1>, <attrName2> [, <attrNameN>]* [,
    <attrNameN+1>*)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<splitStr>	String	The string to split into component pieces. It should contain enough component parts to assign to each of the attribute names listed. Each component part is separated from the others by the delimiter character.	No
<delimiter>	SingleCharacter or Character Sequence	The character(s) used to divide the <splitStr> into its component pieces.	Yes
<formatSpec>	#s[#s]+[#s*]	A format specification may be used in place of a delimiting character. The format specification consists of a series of field widths, separated by s characters. If the last s character is followed by a *, it will be repeated as many times as necessary until the input string is empty. The <splitStr> will be created or broken apart according to the widths specified. The format specification can be distinguished from the delimiting character because the format string must always have more than one character in it.	Yes, although either a delimiter character or a format specification must be present.

Name	Range	Description	Optional
<attrName1>, <attrName2>, <attrNameN>, <attrNameN+1>	String	The names of the attributes assigned the pieces of <splitStr> when it is split apart. The first component of <splitStr> is assigned to the first attribute name, the second component to the second attribute, and so on. The number of attribute names must match the number of component parts of <splitStr>. A repeat attribute is seen as a single attribute for this count. It will be turned into a list attribute. If only one attribute is present, and it is a list attribute, the format specification will instead split the input string into the different indexes of the list. A repeat attribute will continue to enlarge the list as necessary.	At least one attribute name must be present.

Configuration

This function does not accept configuration lines.

Description

The @Split command breaks a string into its component parts and assigns those parts to attributes of a feature. It is used to facilitate translation when a single attribute value in one system implies a set of attribute values in another.

For example, when @Split is invoked on a **destination** line with these parameters:

```
@Split(top|left,|,vertical,horizontal)
```

the `vertical` attribute in the feature is assigned a value of `top`, and the `horizontal` attribute is assigned a value of `left`.

Any character may be chosen to be the delimiter for the input string. However, it is critical that the number of attribute names listed matches the number of component parts present in the input string.

If the delimiter starts with a backslash (“\”), it is interpreted as a quoted special character sequence, as specified in the following table. If the sequence is not listed in the table, then the backslash character is simply ignored.

Sequence	Description
\a	Audible alert (bell) (0x07)

Sequence	Description
\b	Backspace (0x08)
\f	Form feed (0x0c)
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\t	Tab (0x09)
\v	Vertical tab (0x0b)
\,	Comma
\\	Backslash
\/	Slash
\ooo	The digits ooo describe a character in octal notation. (There may be one, two, or three octal digits.)
\xhh	The digits hh describe a character in hexadecimal notation. (There may be one or two hexadecimal digits.)

The delimiter may also be specified in an encoded internal FME format produced by FME Workbench. This encoding allows for a multiple character string with special character handling.

The field attribute names may be names of attribute lists. List attribute names contain empty {} in their names. If the field names are lists, then the input string is assumed to be a list itself, and each element in the input string is broken into pieces and assigned to the next element in the attribute list. The {} are filled in with element numbers starting at 0 as the list is unpacked.

Tip

If one attribute is a list, then they all must be.

Inverse Operation

When invoked on a **source** line of a transfer specification, `@split` is invoked in the inverse direction. In this case, it **combines** the values of the attribute names listed into a single string, separated by the delimiter character, and assigns the result to the transfer variable present as the first argument.

For example, if the clause below was invoked on the source line of a transfer specification, the values of the `vertical` and `horizontal` attributes would be joined together, separated by a `|` character, and assigned to the `%allTogether` transfer variable.

```
@Split(%allTogether,|,vertical,horizontal)
```

If the attributes named are lists, then each element in the lists gets joined together and the transfer variable is set to a list of combined values.

Example

In the first example, a single character string field in a Shape file is used to store the justification code of some text features. When translation from SAIF to Shape occurs, the values of the SAIF text alignment attributes are concatenated together, separated by a |, and assigned to the %justification transfer variable. The JUST attribute of the Shape feature then receives this value. If the vertical alignment value in the SAIF feature originally was TOP, and the horizontal value was LEFT, then JUST would end up being set to TOP|LEFT.

When translation from Shape to SAIF occurs, the %justification transfer variable is assigned the value of the Shape feature's JUST attribute. The @Split command on the SAIF line, which is the *destination* portion of the transfer specification, is run in the forward direction and breaks the %justification value into pieces, assigning them to the text alignment attributes of the SAIF text feature. If the JUST attribute in the Shape feature was originally BOTTOM|RIGHT, then the vertical alignment value in the SAIF feature will be BOTTOM, and the horizontal value will be RIGHT.

```
SAIF Text::TRIM @Split (%justification,|, \
                        textOrSymbol.alignment.vertical, \
                        textOrSymbol.alignment.horizontal)
SHAPE text JUST %justification
```

In the second example, this is extended slightly to consider translation from a SAIF file to a Design file. In a Design file, a single numeric code is used to encode the horizontal and vertical text alignments. However, in SAIF these are separated into two fields. To accommodate this translation, the @Lookup function is used in conjunction with @Split.

```
Lookup JustLUT top|left 1 bottom|right 2
SAIF Text::TRIM @Split (%justification,|, \
                        textOrSymbol.alignment.vertical, \
                        textOrSymbol.alignment.horizontal)
IGDS 23 igds_type igds_text ... \
      igds_justification @Lookup(JustLUT,%justification)
```

When translation is done from SAIF to a Design file, the @Split runs in reverse and glues together the values of the vertical and horizontal text alignment attributes in the SAIF feature. The resulting string is placed into the %justification transfer variable. To fill in the attributes for the Design feature, the @Lookup runs in the forward direction because it is on a destination line. It looks up the value of the %justification transfer variable, for example

`bottom|right`, in the `JustLUT`, and returns the result, for example 2, which is stored in `igds_justification`.

Tip

`@Split` is often used in conjunction with `@Lookup` to decode a numeric code into its components.

When translation is done from Design to SAIF, things work as before but in reverse. The Design file line is the *source*, so the `@Lookup` runs in reverse. It looks up the value of the `igds_justification` on the right-hand side of the `JustLUT`, and places `bottom|right` in the `%justification` transfer variable. Once the new SAIF feature has been created, the `@Split` runs in the forward direction. It splits the value of `%justification`, which is `bottom|right`, into two pieces assigning them to the `vertical` and `horizontal` attributes of the SAIF feature.

The third example shows the use of `@Split` in conjunction with attribute lists. This situation occurs when dealing with multi-line text data. This example works in the same manner as the second example detailed above, except the transfer variable contains a list. The `@Split` and `@Lookup` operate on this list, instead of just a single value.

```
Lookup JustLUT top|left 1 bottom|right 2

SAIF Text::TRIM textOrSymbol.Class TextMultiLine \
    @Split(%justification,|, \
        textOrSymbol.textLines{}.alignment.vertical, \
        textOrSymbol.textLines{}.alignment.horizontal)

IGDS 23 igds_type igds_multi_text ... \
    elements{}.igds_justification
    @Lookup(JustLUT,%justification)
```

The fourth example shows the use of `@Split` in conjunction with date fields. In this case, it is used to concatenate the year, month, and day together into a single string. This is needed because the original SAIF data had them in separate fields but the destination Shape file requires they be placed in a single field. When translation from Shape back to SAIF occurs, the split command breaks the `yyyymmdd` data held in the Shape attribute into the SAIF year, month, and day.

```
SAIF ForestStand::MOF position.geometry.Class BoundedArea \
    @Split (%yyyymmdd,4s2s2s, \
        measurementDate.year, \
        measurementDate.month, \
        measurementDate.day)

SHAPE stands MEASDATE %yyyymmdd
```

@SQL

```
@SQL(<connectionID>,<sqlQuery>[,<resultList>{}])
```

Function Type: Attribute or Feature

Arguments:

Name	Range	Description	Optional
<connectionID>	String	Name of database connection on which to execute query.	No
<sqlQuery>	String	Fully-formed SQL query to pose to database.	No
<resultList>{}	String	An attribute list to hold the results of the query, if there are any.	Yes

Configuration

The @SQL function accepts the following configuration line:

```
SQL <connectionID> [<keyword> <value>]+
```

Keyword	Range	Description	Optional
SERVER_TYPE	String	The type of server being used. For this FME release, the valid values are ODBC, ORACLE, ORACLE8I, EXCEL, MDB, and MSSQL_ADO. When an MDB or Excel server is used, the server name is the name of the MDB or Excel file.	No
SERVER_NAME	String	The name of the server hosting the database. For ODBC servers, this is the ODBC name of the data source. For ORACLE and ORACLE8I servers, this is the SQLNet service name.	No
USER_NAME	String	The user name that accesses the database. Some database sources ignore this value.	Yes
PASSWORD	String	The password that accesses the database. Some database sources ignore this value.	Yes
USE_SSPI	YES or NO	For SQL Server, set this to YES to use Windows Authentication instead of username and password.	Yes

The `SQL` configuration line defines a database connection for a future `@SQL` call. Every invocation of `@SQL` references a connection to a data source.

Some parameters are optional as far as the `SQL` configuration line is concerned. For example, ODBC may require only the `SERVER_NAME` parameter unless a user name and password were assigned to the data source, in which case the `USER_NAME` and `PASSWORD` parameters would also be required.

This function enables the FME to execute arbitrary database commands through a connection to the database. Currently ODBC, MDB, Excel, ORACLE and any FME Objects accessible databases are supported. FME Objects enabled databases include PostGIS/PostgreSQL, SDE 3.x, GeoMedia SQL Server, DB2, and SQL Server.

The purpose of the command is to provide access to Structured Query Language (SQL) statements which perform database updates or queries. An SQL query can result in zero or more results being returned. For example, a query to remove a table does not return any results, but a query to find rows of a table matching a certain set of criteria could return many results.

The `<resultList>` parameter of the `@SQL` call specifies an attribute list to hold the results of the query, if there are any. It must be specified as a proper attribute list name, complete with a trailing `}`. The results are stored on the feature as attributes in the form `<listName>{n}.<attrName>`, where `n` is the result number, starting at 0, and `<attrName>` is the name of a column in the returned table.

It is possible for an SQL query to return results from multiple tables. When writing results to the feature's attributes, a special attribute named `<list-Name>{n}.SQLResultSetTableNum` is created to specify the index of the returned table, starting at 0, for result `n`. Normally, this attribute always has a value of 0. This situation may occur, for example, if an update triggers queries on several tables.

If the `<resultList>` is not specified, then any query results will be stored directly on the feature. The SQL column name is used as the FME attribute name. If multiple rows are returned, the values from the last row will be stored on the feature.

If the query returns any values of a date type, they are stored in the feature in `YYYYMMDD` format. In addition, they are also stored in an attribute with a `.full` suffix in the format `YYYYMMDDHHMMSS`.

Tip

When performing an SQL query on an Excel database, the table name must be suffixed with a `'$'` and surrounded with square brackets in order for the query to execute properly. This is due to limitations of the Excel ODBC driver and how it handles system tables.

Error Handling

If an error is encountered while performing an SQL query, it will be logged to the log file. If `@SQL` is being used as an attribute function, its return value will be the same error message. `@SQL`'s return value will be an empty string when no error occurs.

Inverse Operation

This function performs the same operation in the inverse direction.

Example 1

The following example indicates how an SQL statement is applied to modify a database using the `@SQL` function. The following statement executes the `drop table sharks` SQL statement. The command is executed on an ODBC data source named `sea_world`. No user name or password is needed to access the database.

This example does not expect any results, so no `<resultList>` parameter is specified in the `@SQL` invocation.

```
SQL sampleConn SERVER_TYPE ODBC SERVER_NAME sea_world

FACTORY_DEF * CreationFactory \
OUTPUT FEATURE_TYPE Dropper \
    @SQL(sampleConn,"drop table sharks") \
SAMPLE_RATE 1
```

Example 2

The following example indicates how an SQL statement is executed using the @SQL function. The statement is executed on an Oracle database accessed via the service name `sea_world` with a user identifier of `flipper` and password of `fishypassword`.

This example places results in the attribute list `sqlResults{}`, so the feature ends up with attributes something like this:

Attribute	Value
sqlResults{0}.name	Shamu
sqlResults{0}.animal_type	whale
sqlResults{0}.popularity	100
sqlResults{0}.home_pool	12
sqlResults{0}.building_name	Penguin Gymnastics
sqlResults{0}.building_id	4
sqlResults{1}.name	Mishka
sqlResults{1}.animal_type	seal
sqlResults{1}.popularity	92
sqlResults{1}.home_pool	11
sqlResults{1}.building_name	Penguin Gymnastics
sqlResults{1}.building_id	4

```
SQL sampleConn SERVER_TYPE ORACLE SERVER_NAME sea_world \
    USER_NAME flipper PASSWORD fishypassword
FACTORY_DEF * SamplingFactory \
    INPUT FEATURE_TYPE * \
        @SQL(sampleConn,"SELECT * FROM entertainers,shows \
WHERE popularity > 78",sqlResults{}) \
    SAMPLE_RATE 1
```

If the same query were executed without the attribute list `sqlResults{}`, the feature will end up with attributes something like this. Attributes from the last row selected would be added to the feature:

Attribute	Value
name	Mishka
animal_type	seal
popularity	92
home_pool	11
building_name	Penguin Gymnastics
building_id	4

@SubsetRaster

```
@SubsetRaster(<start row>, <num rows>, <start column>,  
<num columns>)  
  
@SubsetRaster(<start row>, <num rows>, <start column>,  
<num columns>, <num top padding rows>, <num left padding  
columns>, <num bottom padding rows>, <num right padding  
columns>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<start row>	unsigned integers	The starting row from which the subset will be taken.	No
<num rows>	positive integers	The number of rows taken from the raster.	No
<start column>	unsigned integers	The starting column from which the subset will be taken.	No
<num columns>	positive integers	The number of columns taken from the raster.	No
<num top padding rows>	unsigned integers	The number of rows of padding added to the top of the subset raster.	Yes
<num left padding columns>	unsigned integers	The number of columns of padding added to the left of the subset raster.	Yes
<num bottom padding rows>	unsigned integers	The number of rows of padding added to the bottom of the subset raster.	Yes
<num right padding columns>	unsigned integers	The number of columns of padding added to the right of the subset raster.	Yes

Configuration

This function does not accept configuration lines.

Description

The `@SubsetRaster` function is used to reduce a raster to a subset of its original size. This is essentially a clipping operation using pixel bounds instead of ground coordinates.

Padding parameters may optionally be specified in order to add rows and columns of padding around the subset portion of the raster.

This function accepts only features that have raster geometry and is unaffected by raster band and/or palette subselection.

Inverse Operation

The function does nothing when invoked in the inverse direction.

Example

In this example, the `@SubsetRaster` function reduces the input raster to 600 rows and 800 columns, starting from row 84 and column 112 of the initial raster.

```
FACTORY_DEF * SamplingFactory          \
SAMPLE_RATE 1                          \
INPUT FEATURE_TYPE *                   \
    @SubsetRaster(84, 600, 112, 800)    \
OUTPUT FEATURE_TYPE *
```

This is like the previous example, except now 84 rows of padding are added to both the top and bottom of the raster, and 112 columns of padding are added to the left and right of the raster. The resulting raster will have an overall size of 768 rows by 1024 columns.

```
FACTORY_DEF * SamplingFactory          \
SAMPLE_RATE 1                          \
INPUT FEATURE_TYPE *                   \
    @SubsetRaster(84, 600, 112, 800, 84, 112, 84, 112) \
OUTPUT FEATURE_TYPE *
```


@SupplyAttributes

```
@SupplyAttributes()  
@SupplyAttributes(<packedAttributes>)  
@SupplyAttributes(<attrWithPackedAttributes>)  
@SupplyAttributes(<attrName>,<attrValue>  
    [,<attrName>,<attrValue>] *)  
@SupplyAttributes("?",<attrName>[,<attrName>] *  
    ,<attrValue>)  
Function Type: Attribute or Feature
```

Arguments:

Name	Range	Description	Optional
<attrName>	String	The name of the attribute data is being supplied to.	Yes
<attrValue>	String	The value to be assigned to the attribute.	Yes
<packedAttributes>	String	A single special format string that contains numerous attribute names and values packed together. These attributes names and values are unpacked and supplied to the feature.	Yes
<attrWithPackedAttributes>	String	The name of an attribute that contains a packed attribute string.	Yes

Configuration

This function does not accept configuration lines.

Description

This function may be used as either an attribute value function or a feature function. When used as an attribute value function, it takes no parameters. In this case, it *packs* all attribute names and values into one string and returns this string.

When used as a feature function and passed a single parameter, the parameter can either contain the name of an existing attribute whose value is a *packed* list of attribute names and values (as returned by the attribute value function), or a string that is a *packed* list of attribute names and values of the type returned by the attribute value function. It unpacks this string and assigns the specified attribute names in the feature, along with the values provided.

When used as a feature function with an argument list that begins with a question mark as the first argument, followed by one or more attribute names, and terminated by an attribute value, this function performs a conditional assignment on the last attribute name in the list. More specifically:

```
@SupplyAttributes("?", A1, A2, ..., AN, value)
```

will result in value being assigned to AN only if all attributes A1 through AN do not exist on the feature.

Inverse Operation

The inverse of this function is the same as its operation in the forward direction.

Example

When executed in the forward direction, the `SupplyAttributes` function assigns values to the attributes `mif_type` and `mif_pen_width`.

```
SDE roads
MIF roads @SupplyAttributes(mif_type, polyline,
mif_pen_width, 1)
```

In this second example, the `@SupplyAttributes` function is used in conjunction with the value-of operator (`&`) to copy the value of the `standId` attribute to the `nodeNum` attribute.

```
FACTORY_DEF SAIF TeeFactory \
INPUT FEATURE_TYPE ForestStand::MOF \
OUTPUT FEATURE_TYPE * \
@SupplyAttributes(nodeNum, &standId)
```

@System

@System (<command>)

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<command>	String	The command which is to be executed by FME. The command may be any command capable of being run from the host operating system's command line.	No

Configuration

This function does not accept configuration lines.

Description

This function enables the FME to execute any command from within an FME mapping file. The command to run is specified in <command>. The return value from the <command> is returned by the function and can be assigned to an attribute.

The FME translation is normally blocked while the command executes. If the command ends with an "&" character, it is executed in the background. In this case, the return value will be 0 if the command launches successfully, or non-zero otherwise.

The C runtime library `system` function is used to execute the command line passed into @System.

Inverse Operation

This function performs the same operation in the inverse direction.

Example

The following example combines the @System() and @File functions. @System is used to compress a file using the zip command, then @File is used to read the zip file into an attribute named DRAWING_DATA. This is particularly useful when the destination system is a database supporting blob-type columns, because then it is possible for the database to store any type of associated data.

```
FACTORY_DEF * SamplingFactory                                \  
  INPUT FEATURE_TYPE *                                       \  
    @System("zip $(destFILE) c:\tmp\descript.doc")          \  
    @File(DestReadSrcWrite,DRAWING_DATA,$(destFILE))        \  
  SAMPLE_RATE 1
```

@TCL

Note: This function is deprecated. Use @Tcl2 instead.

```
@TCL(<Tcl expression>)
```

Function Type: Attribute or Feature

Arguments:

Name	Range	Description	Optional
<TCL expression>	String	A Tool Command Language (Tcl) expression to execute on the feature. If the expression contains spaces or parentheses, it should be enclosed in quotation marks.	No

Configuration

The @TCL function accepts the following configuration line:

```
TCL <Tcl expression>
```

Name	Range	Description	Optional
<TCL expression>	String	A Tcl expression to execute before the translation begins.	No

Any number of Tool Command Language (Tcl) configuration lines may be present in the mapping file. They are executed by the Tcl interpreter in the order in which they appear before the translation begins. Tcl procedure definitions may be placed on these lines:

```
TCL proc toUpper {value} { return [string toUpper value] }
}
```

As an arbitrary Tcl expression may be placed on these configuration lines, this can be used to effect any action before a translation begins, such as deleting a file:

```
TCL unlink $(FME_MF_DIR)/status.txt
```

The typical use of the configuration line is to source a file containing Tcl procedure definitions which defines procedures to be called later by the @TCL function:

```
TCL source $(FME_MF_DIR)/tclProcs.tcl
```

Note that an absolute path name is required to reliably source externally-defined Tcl procedures. Usually, the `FME_HOME` or `FME_MF_DIR` macros are placed before the file name to fully qualify the path name.

If the file and path name contain spaces, enclose the path in parentheses:

```
TCL source { $(FME_MF_DIR)/tclProcs.tcl }
```

Description

This function evaluates an arbitrary Tcl 8.4.12 expression, and returns the result. Tcl, and its documentation, is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., and other parties. However, the Tcl authors have granted permission to any party to reuse and modify the code and documentation, provided the original copyright holders are acknowledged.

This function allows access to a complete procedural language with rich built-in primitives from within FME mapping files. Tcl procedures can be written to simplify tasks that are awkward to accomplish using standard mapping file syntax, and to allow users to extend FME to solve new problems.

To effectively use this function, familiarity with the Tcl language is required. There are several sources of information available to assist in learning Tcl. The Internet has many Tcl sites — the official site is at <http://www.scriptics.com>. The book *Tcl and the Tk Toolkit* by John Ousterhout, published by Addison-Wesley, ISBN 0-201-63337-X provides a thorough introduction to Tcl.

Note The FME parser assumes that any words beginning with a % (percent) sign are transfer variables when found in a transfer specification. For this reason, it is not possible to pass a % as an argument to a call to @TCL when the call is made in a correlation line. (Note that it is okay to use a % when the call is made on a Factory input or output.) The workaround is to define a Tcl variable which holds the % sign on a TCL configuration line, and then use it in the @TCL call:

```
TCL set formatString %0.5d
...
... @Tcl("format $formatString $FME_Attributes(a)")
```

Global Variables

Three Tcl global variables provide a gateway between an FME feature and the Tcl script: `FME_Attributes`, `FME_FeatureType`, `FME_CoordSys`, `FME_AttrEncoding`, and `FME_AttrNameEncoding` variables. In order to use these in a Tcl procedure, they must be declared as global; otherwise, the variables will be considered to be local and will not affect or interact with the FME feature upon which the function is run. To use them in a function, use a pattern like:

```
proc doSomething {
```

```

global FME_Attributes
global FME_FeatureType
global FME_CoordSys
# rest of function goes here...
}

```

FME_Attributes Array

Within any Tcl procedure called by @TCL, the Tcl global array variable `FME_Attributes` mirrors all attributes of the feature that @TCL was invoked upon. (Note that the `FME_Attributes` variable is completely unrelated to the FME @GlobalVariable function.) These may be read, written, or removed. For example,

```
set FME_Attributes(attr1) 30
```

sets an attribute called `attr1` in the current feature to have the value 30 in the same way that @SupplyAttributes(attr1,30) does.

Similarly,

```
puts $FME_Attributes(attr1)
```

outputs the value of the current feature's `attr1` attribute to standard output.

As well,

```
unset FME_Attributes(attr1)
```

removes the attribute `attr1` from the feature in the same way @RemoveAttribute(attr1) does.

Finally,

```
[array names FME_Attributes]
```

returns a list of all the attributes of the current feature.

Note Because of the interaction between Tcl and FME, you should not use the Tcl command:

```
[info exists FME_Attributes(someAttr)]
```

to determine whether or not an attribute exists on a feature. The above command will always return true. Instead, use this idiom to check whether or not a feature has an attribute:

```
set attrIndex [lsearch [array names FME_Attributes] someAttr]
```

then check that `attrIndex` is not equal to -1.

FME_FeatureType Variable

This global variable is set up to mirror the feature type of the feature. As with the `FME_Attributes` array, it can be set, read, and unset:

```
set FME_FeatureType "Roadster"  
puts $FME_FeatureType
```

FME_CoordSys Variable

This global variable is set up to mirror the coordinate system of the feature. As with the `FME_FeatureType` variable, it can be set, read, and unset:

```
set FME_CoordSys UTM10-83  
puts $FME_CoordSys  
unset FME_CoordSys
```

FME_AttrEncoding Variable

This global variable defines the name of the character encoding in which Tcl assumes a feature's attributes are specified when working with `FME_Attributes`. As Tcl works internally with UTF-8, this means that it will convert attribute values from the given encoding to UTF-8 when evaluating `$FME_Attributes(attrName)`, and will convert values from UTF-8 to the given encoding when setting `FME_Attributes(attrName)`.

The default attribute encoding is the same as the default character encoding of the system on which FME is executing, and might be overridden by Tcl's built-in "encoding system" command. A complete list of supported encodings may be obtained by executing the command "encoding names".

The variable may be set, read, or unset:

```
set FME_AttrEncoding iso8859-2  
puts $FME_AttrEncoding  
unset FME_AttrEncoding
```

If `FME_AttrEncoding` is unset, or is set to be an empty string, it reverts to the default system encoding.

FME_AttrNameEncoding Variable

This global variable defines the name of the character encoding in which Tcl assumes a feature's attributes' names are specified when working with `FME_Attributes`. As Tcl works internally with UTF-8, this means that it will convert attribute names from UTF-8 to the given encoding system, and use the converted name when referring to the actual feature's attributes.

The default attribute encoding is the same as the default character encoding of the system on which FME is executing, and might be overridden by Tcl's built-in "encoding system" command. A complete list of supported encodings may be obtained by executing the command "encoding names".

The variable may be set, read, or unset:

FME_LogMessage Function

The `FME_LogMessage` function is used to write messages to the FME log file. It may be invoked in one of two ways:

```
FME_LogMessage <severity> <messageNumber> [<arg1> ...
<argN>]+
```

or

```
FME_LogMessage <severity> <message>
```

`<severity>` can have one of these values: `fme_inform`, `fme_warn`, `fme_error`, `fme_fatal`, `fme_statistic`, and `fme_statusreport`

When the first form is used, the message number must be present in a file in the `messages` subdirectory under the FME installation directory. The remaining parameters are used to fill in any `%0`, `%1`, ... `%n` parameter holders in the message. For example, if the message was:

```
3011, Opening file %0 for mode %1
```

then `FME_LogMessage` could be called like this:

```
FME_LogMessage fme_inform 3011 /tmp/cacher.txt read
```

In the second form, the message is output directly to the log file.

FME_Execute function

This function is used to call any FME function from within a Tcl procedure. This allows standard FME functions to be called iteratively on the same feature.

The syntax for `FME_Execute` is:

```
FME_Execute <functionName> [<arg1> ... <argN>]+
```

`<functionName>` is the name of any FME function, without the leading `@` sign. Any remaining arguments are passed to the function. For example, the `@Generalize` function is invoked like this:

```
FME_Execute Generalize Douglas 10
```

and this would accomplish the same effect as if `@Generalize(Douglas,10)` were invoked elsewhere in the mapping file.

If the function that is invoked causes the feature to be deleted, the translation will be aborted. The `@Generalize` function may do this if the feature's total length is less than the tolerance value passed in.

Inverse Operation

This function has no inverse.

Example

In the example below, a Tcl procedure is used to calculate the boundaries of a section of land given an FME feature that has the boundaries of its containing township. The Tcl procedure `sectionBounds` is defined in the file `sections.tcl` that resides in the same directory as the calling mapping file. Prior to running the mapping file, the `sections.tcl` was debugged by sourcing it interactively in the Tcl shell (`tclsh`) and calling it from there.

```
tclsh
source sections.tcl
# Define a stub for the FME_Coordinates function call
proc FME_Coordinates { args } { puts "calling coords" }
# Set up some values in the global FME_Attributes array
set FME_Attributes(townshipWidth) 6
set FME_Attributes(townshipHeight) 12
set FME_Attributes(townshipMinY) 100
set FME_Attributes(townshipMinX) 400
# Now call our procedure
sectionBounds 1
# And see if we made any change
puts $FME_attributes(sectionMinX)
The sections.tcl file contains this TCL source code:
# -----
# Compute the bounds of the section, which is passed in to
# the function as its only argument.
# This assumes that the global variable FME_Attributes already has
# entries in it for the containing township bounds (townshipMinX,
# townshipMinY, townshipHeight, townshipWidth)

# The section grid looks like this:
# 31 32 33 34 35 36    ROW 5
# 30 29 28 27 26 25    ROW 4
# 19 20 21 22 23 24    ROW 3
# 18 17 16 15 14 13    ROW 2
# 7  8  9 10 11 12    ROW 1
# 6  5  4  3  2  1    ROW 0
#
# 0  1  2  3  4  5    COLUMN

proc sectionBounds {section} {
    # First secure access to the global array holding the FME
    # feature attributes
    global FME_Attributes

    # First determine the size of a section in the township
    set sectionWidth [expr $FME_Attributes(townshipWidth) / 6.0]
```

```

set sectionHeight [expr $FME_Attributes(townshipHeight) / 6.0]

# Now figure out which row and column of the section in the township.
set row [expr int( ($section - 1) / 6)]
set col [expr int( ($section - 1) % 6)]

# Adjust for even rows, when the numbering goes the wrong way
if { [expr ($row % 2)] == 0 } {set col [expr 5 - $col]}

# Now calculate the min and max corners for the section
set minX [expr $col * $sectionWidth + $FME_Attributes(townshipMinX)]
set maxX [expr $minX + $sectionWidth]
set minY [expr $row * $sectionHeight + $FME_Attributes(townshipMinY)]
set maxY [expr $minY + $sectionHeight]

# And set them as attributes in the FME feature
set FME_Attributes(sectionMinX) $minX
set FME_Attributes(sectionMaxX) $maxX
set FME_Attributes(sectionMinY) $minY
set FME_Attributes(sectionMaxY) $maxY
set FME_Attributes(sectionHeight) $sectionHeight
set FME_Attributes(sectionWidth) $sectionWidth

# And finally set the coordinates of the feature to these bounds
FME_Coordinates dimension 2
FME_Coordinates resetCoords
FME_Coordinates addCoord $minX $minY
FME_Coordinates addCoord $minX $maxY
FME_Coordinates addCoord $maxX $maxY
FME_Coordinates addCoord $maxX $minY
FME_Coordinates addCoord $minX $minY
FME_Coordinates geomType fme_polygon
}

```

An example of a mapping file that tests the function is given here.

```

# This is a simple little mapping file that tests the @TCL function.
# At startup time the Tcl procedures in the "sections.tcl"
# file are loaded via the Tcl source command.
# Then a single feature is created with the creation factory, some
# attributes are added to it, and the Tcl function is invoked.

TCL source "$(FME_MF_DIR)/sections.tcl"

READER_TYPE NULL
WRITER_TYPE NULL
NULL_DATASET null

```

```

FACTORY_DEF * CreationFactory                                \
    OUTPUT FEATURE_TYPE sectionFeature                      \
        @SupplyAttributes(townshipWidth,6)                  \
        @SupplyAttributes(townshipHeight,12)                \
        @SupplyAttributes(townshipMinX,400)                  \
        @SupplyAttributes(townshipMinY,100)                  \
        @SupplyAttributes(section,6)                         \
        @Log("Before")                                       \
        @TCL("sectionBounds &section")                       \
        @Log("After")

```

After the @TCL function is called, the feature will have the various section attributes present on it (sectionMaxY, sectionMinY, sectionMaxX, sectionMinX, sectionWidth, sectionHeight). Its geometry will also be set to the section bounds polygon.

@Tcl2

@Tcl2(<Tcl expression>)

Note @Tcl2 (note the lowercase "cl") has better performance than @TCL because it does not make available the FME_Attributes global array variable, which is expensive to set up. Functions are provided to access feature attributes and should be used instead of the FME_Attributes array. @TCL is now deprecated as a function and its use is discouraged.

The FME_AttributeNames function is available in both @TCL and @Tcl2.

Function Type: Attribute *or* Feature

Arguments:

Name	Range	Description	Optional
<TCL expression>	String	A Tool Command Language (Tcl) expression to execute on the feature. If the expression contains spaces or parentheses, it should be enclosed in quotation marks.	No

Configuration

The @Tcl2 function accepts the following configuration lines:

```
Tcl2 <Tcl expression>
Tcl2 FME_Decode <encoded Tcl expression>
```

Name	Range	Description	Optional
<Tcl expression>	String	A Tcl expression to execute before the translation begins.	No
<encoded Tcl expression>	Encoded string	Same as above, but the string is encoded in an internal FME format, as produced by Workbench's TEXT_EDIT_TCL transformer parameter type.	

Any number of Tool Command Language (Tcl) configuration lines may be present in the mapping file. They are executed by the Tcl interpreter in the order in which they appear before the translation begins. Tcl procedure definitions may be placed on these lines:

```
Tcl2 proc toUpper {value} { return [string toUpper value] }
```

Note that since the Tcl interpreter compiles procedures, it is much more efficient to make small procedures and call them from @Tcl2 than it is to put the equivalent Tcl code inline in the @Tcl2 arguments directly.

As an arbitrary Tcl expression may be placed on these configuration lines, this can be used to effect any action before a translation begins, such as deleting a file:

```
Tcl2 unlink $(FME_MF_DIR)/status.txt
```

The typical use of the configuration line is to source a file containing Tcl procedure definitions which defines procedures to be called later by the @Tcl2 function:

```
Tcl2 source $(FME_MF_DIR)/tclProcs.tcl
```

Note that an absolute path name is required to reliably source externally-defined Tcl procedures. Usually, the FME_HOME or FME_MF_DIR macros are placed before the file name to fully qualify the path name.

If the file and path name contain spaces, enclose the path in parentheses:

```
Tcl2 source {$(FME_MF_DIR)/tclProcs.tcl}
```

Note In some situations, it is useful to run an arbitrary Tcl script just before and/or just after a translation completes. In these scenarios, the FME_BEGIN_TCL and FME_END_TCL directives can be used. See the documentation on Translation Begin/End Hooks in the FME Configuration section of the FME Fundamentals manual (available in the Online Documentation area of Safe Software's website or through FME help menus).

Description

This function evaluates an arbitrary Tcl 8.5.2 expression, and returns the result. Tcl, and its documentation, is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., and other parties. However, the Tcl authors have granted permission to any party to reuse and modify the code and documentation, provided the original copyright holders are acknowledged.

This function allows access to a complete procedural language with rich built-in primitives from within FME mapping files. Tcl procedures can be written to simplify tasks that are awkward to accomplish using standard mapping file syntax, and to allow users to extend FME to solve new problems.

To effectively use this function, familiarity with the Tcl language is required. There are several sources of information available to assist in learning Tcl. The Internet has many Tcl sites – the official site is at <http://www.scriptics.com>. The

book *Tcl and the Tk Toolkit* by John Ousterhout, published by Addison-Wesley, ISBN 0-201-63337-X provides a thorough introduction to Tcl.

Note The FME parser assumes that any words beginning with a % (percent) sign are transfer variables when found in a transfer specification. For this reason, it is not possible to pass a % as an argument to a call to @Tcl2 when the call is made in a correlation line. (Note that it is okay to use a % when the call is made on a Factory input or output.) The work-around is to define a Tcl variable which holds the % sign on a Tcl2 configuration line, and then use it in the @Tcl2 call:

```
Tcl2 set formatString %0.5d
...
... @Tcl2("format $formatString [FME_GetAttribute a])")
```

FME_AttrEncoding Variable

This global variable defines the name of the character encoding in which Tcl assumes a feature's attributes are specified when working with FME_GetAttribute and FME_SetAttribute functions. As Tcl works internally with UTF-8, this means that it will convert attribute values from the given encoding to UTF-8 when getting the value of attribute with FME_GetAttribute function, and will convert values from UTF-8 to the given encoding when setting the attribute value using FME_SetAttribute function.

The default attribute encoding is the same as the default character encoding of the system on which FME is executing, and might be overridden by Tcl's built-in "encoding system" command. A complete list of supported encodings may be obtained by executing the command "encoding names".

The variable may be set, read, or unset:

```
set FME_AttrEncoding iso8859-2
puts $FME_AttrEncoding
unset FME_AttrEncoding
```

If FME_AttrEncoding is unset, or is set to be an empty string, it reverts to the default system encoding.

FME_AttributeExists function

This function is used to determine if an attribute is present on a feature.

The syntax for FME_AttributeExists is:

```
FME_AttributeExists <attributeName>
```

<attributeName> is the name of the attribute whose value is to be retrieved.

If the attribute was present on the feature, the value 1 is returned; otherwise, 0 is returned.

FME_AttrNameEncoding Variable

This global variable defines the name of the character encoding in which Tcl assumes a feature's attributes' names are specified when working with `FME_GetAttribute` and `FME_SetAttribute` functions. As Tcl works internally with UTF-8, this means that it will convert attribute names from UTF-8 to the given encoding system, and use the converted name when referring to the actual feature's attributes.

The default attribute encoding is the same as the default character encoding of the system on which FME is executing, and might be overridden by Tcl's built-in "encoding system" command. A complete list of supported encodings may be obtained by executing the command "encoding names".

The variable may be set, read, or unset:

```
set FME_AttrNameEncoding iso8859-2
puts $FME_AttrNameEncoding
unset FME_AttrNameEncoding
```

If `FME_AttrNameEncoding` is unset, or is set to be an empty string, it reverts to the default system encoding.

FME_AttributeNames function

This function is used to retrieve the names of all feature attributes and return them as a Tcl list.

The syntax for `FME_AttributeNames` is:

```
FME_AttributeNames
```

FME_CoordSys Variable

This global variable is set up to mirror the coordinate system of the feature. As with the `FME_FeatureType` variable, it can be set, read, and unset:

```
set FME_CoordSys UTM10-83
puts $FME_CoordSys
unset FME_CoordSys
```

FME_Coordinates Function

In any Tcl procedure invoked by @Tcl2, the `FME_Coordinates` function can be used to read and write the coordinates of the feature. It provides several options, as listed below:

Option	Description
<code>dimension (2 3)</code>	Sets the dimension of the feature – either 2 or 3.
<code>numCoords</code>	Returns the number of coordinates in the feature.
<code>resetCoords</code>	Removes all the current coordinates from the feature. Note: Calling <code>resetCoords</code> has no effect on the <code>fme_geometry</code> or <code>fme_type</code> attributes. These must be reset or readjusted explicitly by the script so the feature is not left in an inconsistent state.
<code>getCoord (x y z) <index></code>	Returns the coordinate value for the specified axis at the specified <code><index></code> – <code><index></code> ranges from zero to one less than the number of coordinates.
<code>addCoord <xvalue> <yvalue> [<zvalue>]</code>	Adds the coordinate specified to the end of the feature's geometry.
<code>geomType</code>	Returns the geometry type of the feature – it will be one of the values allowed for the <code>fme_geometry</code> attribute as documented in the <i>FME Architecture</i> section of the <i>FME Fundamentals</i> on-line help file (available in the Workbench Help menu): <div style="margin-left: 20px;"> <code>fme_point</code> <code>fme_line</code> <code>fme_polygon</code> <code>fme_donut</code> <code>fme_aggregate</code> <code>fme_undefined</code> </div> Sets the geometry type of the feature to the passed-in value. No sanity checking is done. It is important that this be done with care since setting an incorrect geometry type can later hinder FME.

FME_CopyAttribute function

This function is used to copy a feature attribute. The attribute's original internal data storage type is preserved by using this function. This can be significant when working with high precision floating point values.

The syntax for `FME_CopyAttribute` is:

```
FME_CopyAttribute <newName> <oldName>
```

If the feature did not contain an attribute with the name `oldName`, then the function does nothing. It will not create a new blank valued attribute.

FME_Execute function

This function is used to call any FME function from within a Tcl procedure. This allows standard FME functions to be called iteratively on the same feature.

The syntax for `FME_Execute` is:

```
FME_Execute <functionName> [<arg1> ... <argN>]+
```

`<functionName>` is the name of any FME function, without the leading @ sign. Any remaining arguments are passed to the function. For example, the `@Generalize` function is invoked like this:

```
FME_Execute Generalize Douglas 10
```

and this would accomplish the same effect as if `@Generalize(Douglas,10)` were invoked elsewhere in the mapping file.

If the function that is invoked causes the feature to be deleted, the translation will be aborted. The `@Generalize` function may do this if the feature's total length is less than the tolerance value passed in.

FME_FeatureType Variable

This global variable is set up to mirror the feature type of the feature. As with the other variables, it can be set, read, and unset:

```
set FME_FeatureType "Roadster"
puts $FME_FeatureType
```

FME_GetAttribute function

This function can be used to get a value of attribute on the feature from within a Tcl procedure.

The syntax for `FME_GetAttribute` is:

```
FME_GetAttribute <attributeName>
```

`<attributeName>` is the name of the attribute whose value is to be retrieved.

If the attribute was not present on the feature, an empty string is returned as the value.

FME_LogMessage Function

The `FME_LogMessage` function is used to write messages to the FME log file. It may be invoked in one of two ways:

```
FME_LogMessage <severity> <messageNumber> [<arg1> ...
```

```
<argN>]+
```

or

```
FME_LogMessage <severity> <message>
```

<severity> can have one of these values: `fme_inform`, `fme_warn`, `fme_error`, `fme_fatal`, `fme_statistic`, and `fme_statusreport`

When the first form is used, the message number must be present in a file in the `messages` subdirectory under the FME installation directory. The remaining parameters are used to fill in any `%0`, `%1`, ... `%n` parameter holders in the message. For example, if the message was:

```
3011, Opening file %0 for mode %1
```

then `FME_LogMessage` could be called like this:

```
FME_LogMessage fme_inform 3011 /tmp/cacher.txt read
```

In the second form, the message is output directly to the log file.

FME_RenameAttribute function

This function is used to rename a feature attribute. The attribute's original internal data storage type is preserved by using this function. This can be significant when working with high precision floating point values.

The syntax for `FME_RenameAttribute` is:

```
FME_RenameAttribute <newName> <oldName>
```

If the feature did not contain an attribute with the name `oldName`, then the function does nothing. It will not create a new blank valued attribute.

FME_SetAttribute function

This function is used to set the value of an attribute on a feature.

The syntax for `FME_SetAttribute` is:

```
FME_SetAttribute <attributeName> <attrValue>
```

<attributeName> is the name of the attribute whose value would be set to <attrValue>.

FME_TempFilename function

This function generates a temporary filename in the FME temporary directory. The filename is guaranteed to be a unique, new file.

Note that FME will create an empty file with the given name; you must delete it when you are done.

The syntax for `FME_TempFilename` is:

```
FME_TempFilename [<prefix>] [<suffix>]
```

If a prefix and suffix are not provided, the filename will be returned as an arbitrarily uniquely named file in the FME temporary directory. (For information on where this directory is located, search “Temporary Directory Determination” in the *FME Fundamentals Reference* help file, which is accessible from the Workbench Help menu.)

If a prefix is provided, then that is used as the beginning portion of the filename within the temporary directory, and the suffix is used as the ending portion. Typically the suffix is used to append an extension, and in this case, it would have to additionally include a period “.”

For example, this call:

```
FME_TempFilename raster .png
```

would return something like:

```
c:/Documents and Settings/username/Local Settings/Temp/
rastera05921.png
```

FME_UnsetAttributes function

This function is used to remove one or more attributes from a feature.

The syntax for `FME_UnsetAttributes` is:

```
FME_UnsetAttributes <attr1> [<attr2> <attr3 ...>]
```

No error will be generated if the feature did not contain any of the attributes listed. `FME_UnsetAttributes` simply ignores any argument that does not specify an attribute on the feature.

Global Variables

Four Tcl global variables provide a gateway between an FME feature and the Tcl script: `FME_FeatureType`, `FME_CoordSys`, `FME_AttrEncoding`, and `FME_AttrNameEncoding` variables. In order to use these in a Tcl procedure, they must be declared as global; otherwise, the variables will be considered to be local and will not affect or interact with the FME feature upon which the function is run. To use them in a function, use a pattern like:

```
proc doSomething {
    global FME_FeatureType
```

```

        global FME_CoordSys
        # rest of function goes here...
    }

```

Inverse Operation

This function has no inverse.

Example

In the example below, a Tcl procedure is used to calculate the boundaries of a section of land given an FME feature that has the boundaries of its containing township. The Tcl procedure `sectionBounds` is defined in the file `sections.tcl` that resides in the same directory as the calling mapping file. Prior to running the mapping file, the `sections.tcl` was debugged by sourcing it interactively in the Tcl shell (`tclsh`) and calling it from there.

```

tclsh
source sections.tcl
# Define a stub for the FME_Coordinates function call
proc FME_Coordinates { args } { puts "calling coords" }
# Set up some values in the global FME_Attributes array
FME_SetAttribute townshipWidth 6
FME_SetAttribute townshipHeight 12
FME_SetAttribute townshipMinY 100
FME_SetAttribute townshipMinX 400

# Now call our procedure
sectionBounds 1
# And see if we made any change
puts [FME_GetAttribute sectionMinX]
The sections.tcl file contains this TCL source code:
# -----
# Compute the bounds of the section, which is passed in to
# the function as its only argument.
# This assumes that the feature already has attribute entries in it for the
#   containing township bounds (townshipMinX,
#   townshipMinY, townshipHeight, townshipWidth)

# The section grid looks like this:
# 31 32 33 34 35 36      ROW 5
# 30 29 28 27 26 25      ROW 4
# 19 20 21 22 23 24      ROW 3
# 18 17 16 15 14 13      ROW 2
#  7  8  9 10 11 12      ROW 1
#  6  5  4  3  2  1      ROW 0
#
#  0  1  2  3  4  5      COLUMN

proc sectionBounds {section} {
    # First determine the size of a section in the township

```

```

set sectionWidth [expr [FME_GetAttribute townshipWidth] / 6.0]
set sectionHeight [expr [FME_GetAttribute townshipHeight] / 6.0]

# Now figure out which row and column of the section in the township.
set row [expr int( ($section - 1) / 6)]
set col [expr int( ($section - 1) % 6)]

# Adjust for even rows, when the numbering goes the wrong way
if { [expr ($row % 2)] == 0 } {set col [expr 5 - $col]}

# Now calculate the min and max corners for the section
set minX [expr $col * $sectionWidth + [FME_GetAttribute townshipMinX]]
set maxX [expr $minX + $sectionWidth]
set minY [expr $col * $sectionHeight + [FME_GetAttribute townshipMinY]]
set maxY [expr $minY + $sectionHeight]

# And set them as attributes in the FME feature
FME_SetAttribute sectionMinX $minX
FME_SetAttribute sectionMaxX $maxX
FME_SetAttribute sectionMinY $minY
FME_SetAttribute sectionMaxY $maxY
FME_SetAttribute sectionHeight $sectionHeight
FME_SetAttribute sectionWidth $sectionWidth

# And finally set the coordinates of the feature to these bounds
FME_Coordinates dimension 2
FME_Coordinates resetCoords
FME_Coordinates addCoord $minX $minY
FME_Coordinates addCoord $minX $maxY
FME_Coordinates addCoord $maxX $maxY
FME_Coordinates addCoord $maxX $minY
FME_Coordinates addCoord $minX $minY
FME_Coordinates geomType fme_polygon
}

```

An example of a mapping file that tests the function is given here.

```

# This is a simple little mapping file that tests the @Tcl2 function.
# At startup time the Tcl procedures in the "sections.tcl"
# file are loaded via the Tcl source command.
# Then a single feature is created with the CreationFactory, some
# attributes are added to it, and the Tcl2 function is invoked.

```

```
Tcl2 source "${FME_MF_DIR}/sections.tcl"
```

```

READER_TYPE NULL
WRITER_TYPE NULL
NULL_DATASET null

```

```

FACTORY_DEF * CreationFactory \
    OUTPUT FEATURE_TYPE sectionFeature \
        @SupplyAttributes(townshipWidth,6) \
        @SupplyAttributes(townshipHeight,12) \
        @SupplyAttributes(townshipMinX,400) \

```

```
@SupplyAttributes(townshipMinY,100)           \  
@SupplyAttributes(section,6)                 \  
@Log("Before")                               \  
@Tcl2("sectionBounds &section")              \  
@Log("After")
```

After the @Tcl2 function is called, the feature will have the various section attributes present on it (sectionMaxY, sectionMinY, sectionMaxX, sectionMinX, sectionWidth, sectionHeight). Its geometry will also be set to the section bounds polygon.

@Timestamp

@Timestamp(<formatString>)

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<formatString>	String	Specifies how the current time is formatted when it is output.	No

Configuration

This function does not accept configuration lines.

Description

This functions returns the current time, formatted according to the specification provided in the `formatString` parameter. Ordinary characters placed in the format string are copied to the output without conversion. Conversion specifiers are introduced by a `^` character and are replaced in the `formatString` as follows:

Character	Replacement
<code>^a</code>	The abbreviated weekday name according to the current locale.
<code>^A</code>	The full weekday name according to the current locale.
<code>^b</code>	The abbreviated month name according to the current locale.
<code>^B</code>	The full month name according to the current locale.
<code>^c</code>	The preferred date and time representation for the current locale.
<code>^d</code>	The day of the month as a decimal number ranging from 00 to 31. Note that leading zeros will be added when the day of the month is less than 10. If you want to suppress the leading zero, enter <code>^#d</code> .
<code>^H</code>	The hour as a decimal number using a 24-hour clock ranging from 00 to 23.
<code>^I</code>	The hour as a decimal number using a 12-hour clock ranging from 01 to 12.
<code>^j</code>	The day of the year as a decimal number ranging from 001 to 366.
<code>^m</code>	The month as a decimal number ranging from 00 to 12. Note that leading zeros will be added when the month is less than 10. If you want to suppress the leading zero, enter <code>^#m</code> .

Character	Replacement
[^] M	The minute as a decimal number.
[^] P	Either a.m. or p.m. according to the given time value, or the corresponding strings for the current locale.
[^] S	The second as a decimal number.
[^] U	The week number of the current year as a decimal number, starting with the first Sunday as the first day of the first week.
[^] W	The week number of the current year as a decimal number, starting with the first Monday as the first day of the first week.
[^] w	The day of the week as a decimal, with Sunday being 0.
[^] x	The preferred date representation for the current locale without the time.
[^] X	The preferred time representation for the current locale without the date.
[^] Y	The year as a decimal number without a century ranging from 00 to 99.
[^] Y	The year as a decimal number including the century.
[^] Z	The time zone or name or abbreviation.

For example,

```
@Timestamp ("^Y^m^d")
```

will return "20060913", if invoked on September 13, 2006, and

```
@Timestamp ("^#m^d^Y")
```

will return "9132006".

Similarly, on the same day,

```
@Timestamp("The date is: ^c")
```

would return The date is: Fri Sep 13 16:32:48 PDT 2006.

Inverse Operation

This function does nothing if invoked in the inverse direction.

Example

The following example adds a timestamp attribute to each feature that flows through the system:

```
FACTORY_DEF IGDS SamplingFactory \
  INPUT FEATURE_TYPE * date @Timestamp("^Y^m^d") \
  SAMPLE_RATE 1
```


@Transform

```
@Transform(<sourceFormat>, <destinationFormat>,
<linkAttributes>)
```

Function Type: Feature

Arguments:

Name	Range	Description	Optional
<sourceFormat>	FME format name	The name of the format from which the feature was read.	No
<destinationFormat>	FME format name	The name of the format that is the destination of the feature.	No
<linkAttributes>	(ALIAS_GEOMETRY PRESERVE_ATTRIBUTES)	Specifies whether the format-specific attributes should be linked to the generic attributes. See below. Default is PRESERVE_ATTRIBUTES.	Yes

Configuration

The @Transform function does not accept configuration lines.

Description

This function transforms a feature by adding the geometry attributes used by the destination format to based on those supplied to the feature by the source format.

For example, the MIF reader uses the special attribute `mif_type` to hold the kind of entity that was read. The DWG uses `autocad_entity` for this purpose. When `@Transform(MIF, DWG)` is invoked, an `autocad_entity` attribute will be added to the feature with the appropriate value corresponding to the value of `mif_type`. If the `mif_type` was `mif_polyline`, then the value for `autocad_entity` will be `autocad_line`. Any additional attributes required by the `autocad_entity` will also be created and given the correct values. If the `mif_type` was `mif_arc`, then the value for `mif_primary_axis` would be added as the value for `autocad_primary_axis`.

A pseudo-format name called `FME_GENERIC` is available to convert from the generic `fme_type` to and from specific format representations. The allowed values for `fme_type`, together with any special attributes used, are described in the FME Universal Translator on-line help files, in the section *FME Architecture*. By default, all FME features have the `FME_GENERIC` attributes

when the mapping file. However, it is possible for operations in the mapping file to lose these attributes.

The `<linkAttributes>` parameter specifies whether the created format-specific attributes will be “linked” to the generic FME attributes they came from. If this parameter is set to `ALIAS_GEOMETRY`, then the attributes will be linked. In this case, if either the format-specific or generic attribute is changed, the other will be changed with it. In addition, if the `fme_type` attribute of the feature is changed, the format-specific type attribute (for example, `igds_type`) will be removed. If the `<linkAttributes>` parameter is set to `PRESERVE_ATTRIBUTES`, no such linking is done. The default value for the parameter is `PRESERVE_ATTRIBUTES`.

Inverse Operation

This function has no inverse.

Example

In the first example below, linear `Pipeline` features are transformed from having Shape specific geometry attributes to `FME_GENERIC` attributes.

```
FACTORY_DEF SHAPE SamplingFactory \
SAMPLE_RATE 1 \
INPUT FEATURE_TYPE Pipeline @Transform(SHAPE,FME_GENERIC)
```

In the second example below, all features are transformed from having MapInfo TAB specific geometry attributes to having AutoCAD attributes.

```
FACTORY_DEF SHAPE SamplingFactory \
SAMPLE_RATE 1 \
INPUT FEATURE_TYPE * @Transform(MAPINFO,DWG)
```

@UUID

@UUID ()

Function Type: Attribute

Arguments: None

Configuration

The @UUID function does not accept configuration lines.

Description

This function generates and returns a universally unique identifier (UUID), which is almost guaranteed to be unique across time and all clients. The identifier is created from a combination of the computer's hardware characteristics, the current time, and a sequence number. Characteristics of the actual feature data are not used to assist in UUID generation.

The UUID is expressed as a string consisting of:

- 8 hexadecimal digits followed by a hyphen
- three groups of 4 hexadecimal digits, each followed by a hyphen
- 12 hexadecimal digits

UUIDs are 36 bytes in size and look like:

```
7672aac8-fa0b-464c-b0b8-3efa9ae9cd86
6689a182-e611-4021-a6e6-3cfb7c5d48dc
```

This function is currently only supported on the following operating systems: Windows (all variants), Linux, Solaris, and Mac OS X. When invoked on other operating systems, it will return an error.

Example

```
FACTORY_DEF * CreationFactory           \
2D_GEOMETRY 0 0 0 25 0 100             \
OUTPUT FEATURE_TYPE TESTER             \
    uuid @UUID()                       \
    @Log("After")
```

@Value

```
@Value(<attrName>)
```

Function Type: Attribute

Arguments:

Name	Range	Description	Optional
<attrName>	Any attribute name	This function returns the value of the attribute <attrName>. If the attribute does not exist, then an empty string is returned.	No

Configuration

This function does not accept configuration lines.

Description

This function returns the value of the attribute `<attrName>` if it exists in the feature. If the attribute does not exist, an empty string is returned.

Inverse Operation

This function has no inverse.

Example

Given the following feature:



Feature Type: Address	
Attribute Name	Value
cityName	Vancouver
Vancouver	Surrey
Country	Canada
Coordinates: 45695.47, 55900.81	

- 1 @Value(cityName) returns Vancouver.
- 2 @Value(@Value(cityName)) returns Surrey, since the inner @Value(cityName) statement returns Vancouver and the outer @Value(Vancouver) is evaluated to Surrey.
- 3 @Value(streetName) returns an empty string since the attribute streetName is not part of the feature.

@XValue

```
@XValue([(<x-value>|<list attribute>)] [, Reset])
```

Function Type: Attribute or Feature

Arguments:

Name	Range	Description	Optional
<x-value>	Real Number	The value of the x coordinate to store in the feature.	Yes
<list attribute>	attribute name containing {} exactly once.	The name of a list attribute that contains coordinates to be stored in the feature.	Yes

Configuration

This function does not accept configuration lines.

Description

This function may be used as either a feature function or an attribute value function. When used as a feature function, the optional `x-value` parameter must be specified. In this case, the `@XValue` function stores the specified value as the x coordinate of the feature.

If a list attribute is specified in place of a value, then all the values in that list are supplied as coordinates to the feature; each coordinate is supplied in the same way as a single value.

If the optional `Reset` parameter is specified, then the coordinates of the feature are cleared before the x value is added. If it is not specified, then the x value is added to the current feature's geometry, either extending a line if the feature was linear or creating a Point-In-Polygon feature out of a polygonal feature.

When used as an attribute value function, the `x-value` parameter is not specified. In this case, `@XValue` returns the value of the first x coordinate of the feature. This value is then stored in the attribute.

Warning: `@XValue()` should generally be used in conjunction with `@YValue()`. If `@XValue()` is used on its own, it is possible to create a feature with a different number of x and y values. See *Example: Setting Only the X Value on a Text Feature* on page 305.

Inverse Operation

If the optional `x-value` parameter is not specified and the `@XValue` function is encountered on the source line of an attribute transfer, it will do nothing.

However, when `@XValue` is used as a feature function—the `x-value` parameter *was* specified as a transfer variable—and it is encountered on a source line, it will extract the value of the first `x` coordinate in the feature and assign this value to the transfer variable passed to it. List attributes are not handled in the inverse.

Example

In the first example, `@XValue`, `@YValue`, and `@ZValue` are used as feature functions in conjunction with the *ElementFactory*. This factory takes a feature that has an attribute list on it – in this example, `textOrSymbol.characters{}` – and outputs a new feature for each element in the list. However, each output feature does not have any coordinates unless they are added by the `@XValue`, `@YValue`, and `@ZValue` functions. Since functions invoked in factory clauses are always executed in the forward direction, these functions are invoked on the *ElementFactory*'s `OUTPUT ELEMENT` line to set each output feature's coordinates from its attribute values.

The SAIF attribute names in this example have been shortened and are not accurate.

```

FACTORY_DEF SAIF ElementFactory                                \
  INPUT FEATURE_TYPE Toponymy::TRIM                            \
    textOrSymbol.Class TextOnCurve                             \
  LIST_NAME textOrSymbol.characters{}                           \
  OUTPUT ELEMENT FEATURE_TYPE Toponymy::TRIM                   \
    textOrSymbol.Class TextFromMulti                           \
    @XValue(&x)                                                  \
    @YValue(&y)                                                  \
    @ZValue(&z)                                                  \

```

In this example, an input feature entering the factory looks like:



Feature Type: Toponymy::TRIM	
Attribute Name	Value
textOrSymbol.Class	TextOnCurve
textOrSymbol.characters{0}.x	5001
textOrSymbol.characters{0}.y	40001
textOrSymbol.characters{0}.z	101
textOrSymbol.characters{0}.t	Hello
ext	

This feature is broken into two output features by the *ElementFactory*. Before the functions are executed, the first output feature looks like:

After the functions are run, the feature looks like:

In the second example, `@XValue` and `@YValue` are used as attribute value functions in conjunction with the `ListFactory`. This factory takes in a number of features and forms their attributes into a giant attribute list on the output feature. The `@XValue` and `@YValue` functions are used in this situation to

extract the values of the x and y coordinates, and supply them as attributes to the feature as it enters the `ListFactory`.

```

FACTORY_DEF SHAPE ListFactory \
  INPUT FEATURE_TYPE tponymy x @XValue() y @YValue() \
  LIST_NAME elements{} \
  OUTPUT LIST FEATURE_TYPE tponymylist

```

In this example, an input feature about to enter the factory looks like:



Feature Type: Toponymy::TRIM	
Attribute Name	Value
textOrSymbol.Class	TextLine
text	Hello
Coordinates: 5001,40001	

Notice the addition of the x and y attributes after the functions are run as the element enters the factory:



Feature Type: Toponymy::TRIM	
Attribute Name	Value
textOrSymbol.Class	TextLine
x	5001
y	40001
text	Hello
Coordinates: 5001,40001	

Example: Setting Only the X Value on a Text Feature

```

FACTORY_DEF * TestFactory \
  FACTORY_NAME "TEXT - Set Horizontal Text Insertion Point" \
  INPUT FEATURE_TYPE * fme_type fme_text \
  TEST &horiz_just = 1 \
  OUTPUT PASSED FEATURE_TYPE * \
  OUTPUT FAILED FEATURE_TYPE * \
  @SupplyAttributes(temp_y,@Coordinate(y,0)) \
  @XValue(@Evaluate(&world_insertion_x - \
    &x_text_multiplier), reset) \
  @YValue(&temp_y)

```

@YValue

```
@YValue([(<y-value>|<list attribute>)])
```

Function Type: Attribute or Feature

Arguments:

Name	Range	Description	Optional
<y-value>	Real Number	The value of the y coordinate to store in the feature.	Yes
<list attribute>	attribute name containing {} exactly once.	The name of a list attribute that contains coordinates to be stored in the feature	Yes

Configuration

This function does not accept configuration lines.

Description

This function is identical in its operation to the @XValue function, except it operates on the y coordinate rather than on the x coordinate. It may be used either as a feature function or an attribute value function. When used as a feature function, the optional y-value parameter must be given. In this case, the @YValue function stores the specified value as the y coordinate of the feature.

If a list attribute is specified in place of a value, then all the values in that list are supplied as coordinates to the feature; each coordinate is supplied in the same way as a single value.

When used as an attribute value function, the y-value parameter is not specified. In this case, @YValue returns the value of the first y coordinate of the feature. This value is then stored in the attribute.

Warning: @YValue() should generally be used in conjunction with @XValue(). If @YValue() is used on its own, it is possible to create a feature with a different number of x and y values. See *Example: Setting Only the Y Value on a Text Feature* on page 307.

Inverse Operation

If the optional y-value parameter is not specified and the @YValue function is encountered on the source line of an attribute transfer, it will do nothing.

However, when @YValue is used as a feature function – the y-value parameter *was* specified as a transfer variable – and it is encountered on a source line, it extracts the value of the first y coordinate in the feature and assigns this value to the transfer variable passed to it. List attributes are not handled in the inverse.

Example

See the examples for @XValue.

Example: Setting Only the Y Value on a Text Feature

```

FACTORY_DEF * TestFactory                                \
  FACTORY_NAME "TEXT - Set Vertical Text Insertion Point" \
  INPUT FEATURE_TYPE * fme_type fme_text                \
  TEST &vert_just = 1                                    \
  OUTPUT PASSED FEATURE_TYPE *                          \
  OUTPUT FAILED FEATURE_TYPE *                          \
    @XValue(@Coordinate(x,0),reset)                      \
    @YValue(@Evaluate(&insertion_y - &y_text_offset))

```

@ZValue

```
@ZValue([(<z-value>|<list attribute>)])
```

Function Type: Attribute or Feature

Arguments:

Name	Range	Description	Optional
<z-value>	Real Number	The value of the z coordinate stored in the feature.	Yes
<list attribute>	attribute name containing {} exactly once.	The name of a list attribute that contains coordinates to be stored in the feature	Yes

Configuration

This function does not accept configuration lines.

Description

The @ZValue function stores the specified value as the z coordinate in the feature. If the feature contains multiple coordinates, then all coordinates will be set to the specified z-value.

If a list attribute is specified in place of a value, then all the values in that list are supplied as successive coordinates to the feature.

When used as an attribute value function, the z-value parameter is not specified. In this case, @ZValue returns the value of the first z coordinate of the feature. This value is then stored in the attribute.

Warning: @ZValue() should generally be used in conjunction with @XValue() and @YValue(). If @ZValue() is used on its own, it is possible to create a feature with a different number of x, y and z values.

Inverse Operation

When executed in the inverse direction on the source line of a transformation specification, the z value of the feature is assigned to the transfer variable passed to the @ZValue function. This allows the same ZValue statement to be used for both directions of a translation. List attributes are not handled in the inverse.

Example

In the following example, when the translation proceeds from Shape to SAIF, the value of the Shape `height` attribute is assigned to the transfer variable `%height`. It is then passed to the `ZValue` function, resulting in the `z` coordinate of the SAIF feature being set to the attribute value of `height`.

In the inverse direction, the `z` coordinate value of the SAIF feature is placed into the transfer variable `%height` which is then stored in the `height` attribute of the outgoing Shape feature.

```
SHAPE roads height %height
SAIF Roads::TRIM @ZValue(%height)
```


FME Factories

FME Factories operate on zero or more FME features, producing zero or more FME features. Because factories can work on more than one feature at a time, their output can be the result of combining the features in a variety of ways. This chapter provides detailed information on each FME factory, and describes the configuration clauses it takes, the inputs it allows, any assumptions it makes about the input data, and the outputs it produces.

AggregateFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> AggregateFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [LIST_NAME <list name>{}]
  [COUNT_ATTRIBUTE <attribute name>]
  [ACCUMULATE_ATTRIBUTES [(yes|no)]]
  [GROUP_BY [<attribute name>]+]*
  [SUM_FIELDS [<attribute name>]+]
  [AVERAGE_FIELDS [<attribute name>]+]
  [WEIGHTED_AVERAGE_FIELDS [<attribute name>]+]
  [BREAK_BEFORE_FIELD_CHANGE [<attribute name>]+]
  [BREAK_AFTER_FIELD_CHANGE [<attribute name>]+]
  [OUTPUT (AGGREGATE|SINGLETON)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory aggregates the geometries of input features together based on attribute values specified by the `GROUP_BY` clause. One feature is output for each group resulting from the `GROUP_BY` clause. If no `GROUP_BY` clause is specified, then all features fall into the same group and a single feature is output. Features are added to the aggregate being built in the order they are received by this factory. If the order of individual features within the resulting aggregate is important, users can first route the features through a `SortingFactory`.

At most one of `BREAK_BEFORE_FIELD_CHANGE` and `BREAK_AFTER_FIELD_CHANGE` can be specified. Whenever the values of the attributes specified by this clause change from one feature to the next, the aggregate currently being constructed is output. In the case of `BREAK_BEFORE_FIELD_CHANGE`, the newly received input feature is included in the next aggregate, and with `BREAK_AFTER_FIELD_CHANGE` it is made part of the aggregate under construction.

The features output from the `AggregateFactory` have the combined attribution of all features combined to make up the aggregate.

The `SUM_FIELDS` clause identifies attributes whose values are to be summed together when constituent features are aggregated.

The `AVERAGE_FIELDS` clause identifies attributes whose values are to be averaged during the aggregation. The `WEIGHTED_AVERAGE_FIELDS` clause identifies attributes whose values are averaged weighted according to the area

of their original feature. Note that non-polygonal features have an area (and therefore a weight) of zero in the weighted average calculation; therefore, if no polygons contribute to a given aggregate, then the weighted average of any attribute will be infinity.

The optional `LIST_NAME` clause is used to associate attributes with each member of the aggregate. When the aggregate is created, all attributes of each feature joining the aggregate are added as members of the specified attribute list. The index in the list corresponds to the index of the feature's geometry in the aggregate.

If the optional `COUNT_ATTRIBUTE` clause is given, a new attribute will be created on any aggregates output, containing the number of features that were combined to form the aggregate.

If the optional `ACCUMULATE_ATTRIBUTES` clause is specified together with the optional `LIST_NAME` clause, then each singleton feature that is output will also have a list created on it holding the attributes of the single feature that it was based on. As well, this clause causes the attributes of each member feature of an aggregate to be merged onto the feature being output, in addition to be accumulated in the list.

This provides functionality very similar to the `ListFactory`. Either the `ElementFactory` or the `DeaggregateFactory` can then be used to extract the attributes from the list.

Tip

When large aggregates are created, the list can require a great deal of memory. In some situations, the `@KeepAttributes` should be used in the `AggregateFactory`'s input clause to reduce the number of attributes that become part of the list.

This factory is similar to the `ListFactory`. The `ListFactory` aggregates attribution, while the `AggregateFactory` aggregates geometry (and possibly also attribution).

Features with geometries aggregated by this factory may be routed to formats that accommodate aggregates for output.

Tip

The `DeaggregateFactory` is used to split aggregates up into their components.

If the `OUTPUT SINGLETON` clause is specified, features that are the only members of their group are output unchanged (but if the `ACCUMULATE_ATTRIBUTES` and `LIST_NAME` clauses were present, singleton will have a list of their attributes added as well). If `OUTPUT SINGLETON` is not specified, an aggregate containing only one geometry will be output for the group.

If the `FME_GEOMETRY_HANDLING` directive is set to “yes” in the mapping file, the aggregated geometry will preserve arcs, ellipses, and text; they will otherwise be converted into points in the aggregated geometry.

Assumptions

None.

Output Tags

The `AggregateFactory` supports the following output tags.

Tag	Description
AGGREGATE	The features that have aggregate geometry. If the <code>SINGLETON</code> tag is specified, then each of these features will have at least two geometries in it.
SINGLETON	Applies to features that are the only members in a group. If this is not specified, then such features are output using the <code>AGGREGATE</code> tag creating a geometric aggregate with only one geometry in it. If an <code>OUTPUT SINGLETON</code> clause is specified, such features are output unchanged.

Example

The following example illustrates the use of the *AggregateFactory* to merge linear contour features together based on their elevation value.

```

FACTORY_DEF SAIF AggregateFactory           \
  INPUT FEATURE_TYPE Contour::TRIM         \
  GROUP_BY position.geometry.value         \
  OUTPUT AGGREGATE FEATURE_TYPE ContourAggregate

```

Tip

Since no `OUTPUT SINGLETON` clause is specified, groups that have only 1 element in them will still be output as an aggregate.

The features below match the input specification of this factory definition and will enter the factory.



Feature Type: Contour::TRIM

Attribute Name	Value
position.geometry.qualifier	definite

The factory merges the above features together, to produce the aggregate feature below. This feature could be output to Shape or SAIF.

Feature Type: ContourAggregate	
Attribute Name	Value
position.geometry.qualifier	definite
position.geometry.value	20

Coordinates:

Line 1: (477553,5360181,20) (477554,5360182,20)

Line 2: (377553,4360181,20) (377554,4360182,20)

The `qualifier` attribute was copied from the first feature that was encountered in the group.

ArcFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> ArcFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [GROUP_BY [<attribute name>+]*
  (END_NODED|VERTEX_NODED)
  [BREAK_ACROSS_GROUPS [(yes|no)]]
  [PRESERVE_ORIENTATION [(yes|no)]]
  [CLOSE_LOOPS [(yes|no)]]
  [LIST_NAME <list attribute name>{}]
  [OUTPUT (LINE|NODES_BEFORE_PSEUDO_REMOVAL|
    NODES_AFTER_PSEUDO_REMOVAL|INVALID_GEOMETRY)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory takes linear or polygonal features and outputs their component lines with all duplicates and pseudo-nodes removed. Linear features leaving the factory are only broken at topologically significant nodes. This factory is useful for determining the arcs required to divide a surface into regions, or to connect arcs that have the same attribute values.

The input features may be grouped based on attribute values specified in the **GROUP_BY** clause. No attributes are carried across from the **INPUT** features to the **OUTPUT** features. However, all **OUTPUT** features have the **GROUP_BY** attributes added to them before being output. If **BREAK_ACROSS_GROUPS** is present, then the factory will consider all nodes from all groups of features when deciding on topologically significant points. If **BREAK_ACROSS_GROUPS** is not specified, then each group will be considered separately.

The **END_NODED** and **VERTEX_NODED** directives tell the factory about the topology of the input features. **END_NODED** indicates that all features begin and end at topologically significant points, and that none of their vertices connect with any other features. **VERTEX_NODED** indicates that every vertex may be topologically significant and must be considered when looking to join features together.

If **CLOSE_LOOPS** is present, then the factory closes any loops that do not share a node with any other lines in the data set. Such loops are output as polygons by the **OUTPUT LINE** clause. **CLOSE_LOOPS** also ensures that each such loop starts and ends at the lower right-most point. If **CLOSE_LOOPS** is not present, then loops will be output in two pieces, broken at arbitrary points.

Note:

- When the `CLOSE_LOOPS` clause is used, the `ArcFactory` is more efficient at forming polygons than the `PolygonFactory` in scenarios where the polygons do not share edges.

If `PRESERVE_ORIENTATION` is present, then the factory will *not* alter the direction of any of the input line segments in order to join them together.

If `PRESERVE_ORIENTATION` is not present, the directionality of each input line may or may not be adjusted as the new lines are formed.

If `LIST_NAME` is given, then a list will be created on each output feature, containing an element for each input feature which contributed to that geometry, in order of appearance.

The factory also enables users to get the unique set of nodes both before and after the `ArcFactory` processing. If the user wants the set of nodes as defined by the input set of arcs, then they are output via the `NODES_BEFORE_PSEUDO_REMOVAL` output tag. If the user wants the set of nodes as defined by the output set of arcs, then they are output via the `NODES_AFTER_PSEUDO_REMOVAL`.

Tip

`END_NODED` does not make sense when polygons are input to the `ArcFactory`.

In the context of this factory, the term *arc* does not refer to a portion of a circle, but rather to a topologically significant linestring

If the `FME_GEOMETRY_HANDLING` directive is set to yes in the mapping file, arcs and ellipses are preserved throughout the processing of the component lines; otherwise they are stroked prior to processing.

Assumptions

This factory assumes that all input lines are already topologically noded. The factory also assumes that the input polygons and arcs have already been processed by a GIS product that has ensured the data is clean. The `ArcFactory` does not provide any snapping or intersection capabilities.

Output Tags

The `ArcFactory` supports the following output tags.

Tag	Description
LINE	All linear features produced by this factory are output according to the <code>OUTPUT</code> clause identified by this tag.
NODES_BEFORE_PSEUDO_REMOVAL	This tag tells the factory to output the node features before removal of the pseudo nodes. Pseudo nodes are those nodes which are not topologically significant. That is, these nodes will be removed by the <i>ArcFactory</i> processing. This results in each arc endpoint being output once. If multiple arcs share an endpoint, then it is output only once. Nodes in which exactly two arcs share the same endpoint are also output.
NODES_AFTER_PSEUDO_REMOVAL	This tag tells the factory to output the node features after removal of the pseudo nodes. These are the nodes which form the end points of the arcs that are output from the factory.
INVALID_GEOMETRY	This tag tells the factory to output any input features that had an invalid geometry, and couldn't be processed.

Example

The following example defines an `ArcFactory` that joins together all road features with the same number of lanes. The factory is activated only when reading from SAIF. Any loops are closed into rings and the orientation of the roads is preserved. When features leave the factory, the only attribute that they will have is `numberOfLanes`.

```

FACTORY_DEF SAIF ArcFactory          \
INPUT FEATURE_TYPE Road::TRIM        \
GROUP_BY numberOfLanes                \
END_NODDED                           \
CLOSE_LOOPS                          \
PRESERVE_ORIENTATION                 \
OUTPUT LINE FEATURE_TYPE Road::TRIM

```

BoundingBoxFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> BoundingBoxFactory
[FACTORY_NAME <factory name>]
[INPUT FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*
[BOUNDING_BOX_TYPE (COORDINATE|GEOMETRIC)]
[GROUP_BY [<attribute name>+]*]
[OUTPUT (ORIGINAL|BOUNDING_BOX)
  FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*

```

Overview

This factory takes a set of point, linear, polygonal, and/or aggregate features, and computes the bounding box, which is 2D and contains all features. The bounding box is defined as the minimum enclosing rectangle for all input features. The minimum rectangle is such that all sides of the rectangle are parallel to the x axis and the y axis.

The input features may be partitioned into groups based on attribute values using the `GROUP_BY` clause and one bounding box feature is output for each group. If the `GROUP_BY` clause is not specified, then all input features will be processed together and a single bounding box will be output.

If the `BOUNDING_BOX_TYPE` clause exists, it specifies the type of bounding box that will be computed. A `BOUNDING_BOX_TYPE` of `COORDINATE` specifies that the bounding box should be the maximum bounds of all the coordinates in the input feature. A `BOUNDING_BOX_TYPE` of `GEOMETRIC` specifies that the bounding box should be the bounding box of the geometry of the feature. The difference between the two options occurs in some point features, like text, ellipse, and arc features. For instance, text features only have a single coordinate, so a coordinate bounding-box would only contain this single point, but a geometric bounding box would contain the entire text feature as if it were rendered with a fixed-width font at its given height.

No attributes are carried across from the `INPUT` features to the `OUTPUT BOUNDING_BOX` features. However, all `OUTPUT BOUNDING_BOX` features are assigned the `GROUP_BY` attributes before being output.

The `OUTPUT ORIGINAL` features are untouched as a result of their visit to the factory. If this clause is not specified, then the original features will be deleted.

This factory differs from the `@Bounds()` function in that the function computes the bounding box of a single feature, whereas this factory computes the bounding box of a group of features.

Assumptions

None.

Output Tags

The `BoundingBoxFactory` supports the following output tags.

Tag	Description
<code>BOUNDING_BOX</code>	The bounding box of the group of features, stored as a simple polygon.
<code>ORIGINAL</code>	The original, unchanged input features.

Example

The following example takes all points from a Shape file named `shotpoints` and groups them by their `CLASS` attribute. A bounding box is output for each group. The original points are not output by the factory.

```

SHAPE_IN_DEF shotpoints \
  SHAPE_GEOMETRY      shape_point \
  CLASS               char(10)
SHAPE_OUT_DEF boxes \
  SHAPE_GEOMETRY      shape_polygon \
  CLASS               char(10)
FACTORY_DEF SHAPE_IN BoundingBoxFactory \
  INPUT FEATURE_TYPE shotpoints \
  GROUP_BY CLASS \
  OUTPUT BOUNDING_BOX FEATURE_TYPE created_boxes \
SHAPE_IN created_boxes \
  CLASS               %class \
SHAPE_OUT boxes \
  CLASS               %class

```


AttributeFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> AttributeFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  ATTR_NAME_LABEL <attribute name label>
  ATTR_VALUE_LABEL <attribute value label>
  EXPLODING_TYPE (EXPLODE_AS_FEATURES|EXPLODE_AS_LIST) ]
  KEEP_GEOEMTRY (YES|NO)
  KEEP_ATTRIBUTES (YES|NO)
  LIST_NAME <list name>{}
  [OUTPUT (BASE|ELEMENT)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory takes a feature with attributes and separates them as new pair of attributes (attribute_name/attribute_value) in individual cloned features, or make a list of it's attributes as a new list attribute in the processed feature.

When EXPLODE_AS_FEATURES mode is specified, the processed feature is cloned once for each attribute. The definition of each initial attribute will be contained by a pair of new attributes : One specified by ATTR_NAME_LABEL with it's associated ATTR_VALUE_LABEL. Initial attributes and geometry is conserved when cloning.

When EXPLODE_AS_LIST mode is specified, the processed feature will have a new list attribute defined by <list name> and will contain (attribute_name/attribute_value) pairs for each attribute of the feature. Initial attributes and geometry are conserved after processing.

When KEEP_GEOMETRY is specified, the geometry on the processed features are conserved. Otherwise, the geometry on the feature is removed.

When KEEP_ATTRIBUTES is specified, the initial attributes on the processed features are conserved. Otherwise, the initial attributes on the feature is removed.

Assumptions

None.

Output Tags

The `AttributeFactory` supports the following output tags.

Tag	Description
EXPLODED	The exploded feature(s).

Example

Suppose we have a `BASE` feature with the following 2 attributes. (The `LIST_NAME` used here would be `"list_name{"`.)

```
Name = John
Type = Employee
```

The directive `EXPLODE_AS_FEATURES` will produce 2 elements with 4 attributes:

```
Name = John
Type = Employee
attribute_name= Name
attribute_value = John

Name = John
Type = Employee
attribute_name= Type
attribute_value = Employee
```

The directive `EXPLODE_AS_LIST` will produce 1 element with 3 attributes:

```
Name = John
Type = Employee
list_name{0}.attribute_name= Name
list_name{0}.attribute_value = John
list_name{1}.attribute_name= Type
list_name{1}.attribute_value = Employee
```

BranchingFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> BranchingFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  TARGET_FACTORY <factory target>
  [FALLBACK_TARGET_FACTORY <fallback factory target>]
  [MAXIMUM_COUNT <maximum loop>]
  [TEST <value> <operator> <value>]
  [OUTPUT (PASSED|FAILED|COUNT_EXCEEDED)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory allows features to be explicitly routed to other factories. If present, the factory evaluates the expression defined by the **TEST** clause. If the result is true or there was no **TEST** clause, then the feature is output via the *PASSED* output tag and sent to the factory named <factory target>. If the evaluation is false, then the feature is output via the *FAILED* output tag and continues through the factory pipeline for regular processing.

If the *PASSED* clause is not specified, then features that passed the test are sent directly to the factory identified by <factory target>. The *PASSED* clause is optional and provides a mechanism to perform operations on features before they are routed to the destination factory.

If the <factory target> could not be found, then the feature will be routed to the <fallback factory target>, if one was specified. If no fallback was specified, or the fallback could not be found, then the factory will cause the translation to fail.

The <factory target> can be specified as a constant name, the value of an attribute present on the feature being routed, or the result of a function call (which will be executed on the feature being routed).

If the *FAILED* clause is not specified, then any features for which the **TEST** clause is false are destroyed.

The **MAXIMUM_COUNT** clause specifies the maximum number of times that a feature is permitted to visit a particular factory. This can be used to guard against infinite loops; however, setting **MAXIMUM_COUNT** to -1 removes the restriction on how many times a feature can go through the factory, thereby allowing infinite loops to occur. If not specified, the default value is 100. Sending the feature backwards through the pipeline is dangerous and should be


```
<instance name>.BranchingFactory.Count
```

The `TEST` clause specifies the expression which determines if a feature is to go out via the `PASSED` output tag or the `FAILED` output tag.

The <value> may be a literal constant, an attribute name preceded by the value of operator (&), or an attribute value function. If it is an attribute value function, then the function will be executed on the current feature and the result will be used for the test.

```
TEST @Area() < 100
TEST &numLanes > 2
TEST "Joe" = "Jerry"
```

Assumptions

None.

Output Tags

Tag	Description
PASSED	Features for which the TEST clause evaluates to true are output here before being sent to the factory specified by the TARGET_FACTORY clause.
FAILED	Features for which the TEST clause evaluates to false are output here before leaving the factory .
COUNT_EXCEEDED	Features that have entered the factory more than the value specified by MAXIMUM_COUNT are output here .

ChoppingFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> ChoppingFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  MAX_VERTICES <maximum vertices>
  [CHOP_POLYGONS] [(yes|no)]
  [OUTPUT (UNTOUCHED|CHOPPED)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory ensures that all features it outputs have less than or equal to <maximum vertices> vertices.

If an input feature has more than this number of vertices, it will be *chopped* into several smaller features. Each new feature will have <maximum vertices> vertices, except for the last one which may have fewer than <maximum vertices> vertices. All new features will have the same attributes as the original feature had and will be output using the CHOPPED clause.

If the feature had <maximum vertices> or fewer vertices, then it is output by the UNTOUCHED clause.

If the CHOP_POLYGONS directive is specified, any polygonal features (possibly with holes), that enter the factory and have more than <maximum vertices> vertices, will be sliced into smaller polygons. The original polygon will be cut by horizontal and vertical lines until each resulting polygon has less than the maximum allowed number of vertices. All polygons produced with this method will have the same attributes as the original polygon before the chopping. If the resulting polygons were dissolved back together, the original polygon would be the result.

Tip

If <maximum vertices> is less than four, the CHOP_POLYGONS directive is ignored.

This factory is used to reduce the number of coordinates in a single feature so that it may later be stored in a system that limits feature sizes.

Assumptions

None.

Output Tags

The ChoppingFactory supports the following output tags.

Tag	Description
UNTOUCHED	All input features with <= MAX_VERTICES vertices are output untouched here.
CHOPPED	Features created by chopping the original feature into smaller pieces are output here. Each feature contains <= MAX_VERTICES vertices, but has all of the original feature's attributes.

Example

In this example, a Shape file containing linear features is converted into a DBF file that holds the feature coordinates, or pair of coordinates, per row. Each feature is assigned a unique ID before it is broken into two point fragments.

The result is a table that looks like this:

Feature ID	X1	Y1	X2	Y2
id0	x0_1	y0_1	x0_2	y0_2
id0	x0_2	y0_2	x0_3	y0_3
id0	x0_3	y0_3	x0_4	y0_4
id1	x1_1	y1_1	x1_2	y1_2
id1	x1_2	y1_2	x1_3	y1_3

```
# -----
# These macros control the 'type' of the feature ID that is
# assigned to each feature as it goes by. They also control the
# method of generation of the feature ID.
# These settings generate sequential numbers, which will be
# unique within this translation. The @GOID function could be
# used to generate globally unique identifiers if this was
# required.
#   MACRO _FeatIdType          number(8,0)
#   MACRO _FeatIdGeneration @Count(FeatureNumber)
# -----
```

```

READER_TYPE SHAPE
WRITER_TYPE TABLE
SHAPE_DATASET .
TABLE_DATASET .
LOG_FILENAME dbf.log
# -----
# Define the input data source
SHAPE_DEF road\
  SHAPE_GEOMETRY      shape_polyline      \
  LENGTH              number(12,3)        \
  ROADTYPE             char(30)            \
# -----
# This table holds the geometry of the roads
TABLE_DEF roadgeom DBF roadgeom.dbf      \
  FEAT_ID  $(_FeatIdType)                \
  X1        number(32,15)                 \
  Y1        number(32,15)                 \
  X2        number(32,15)                 \
  Y2        number(32,15)                 \
# -----
# Now, take the geometry and chop it into features each
# containing two points. As the feature enters the factory it is
# assigned a unique ID. This ID is replicated on each of the
# output CHOPPED features.
FACTORY_DEF SHAPE ChoppingFactory      \
  INPUT FEATURE_TYPE * id $(_FeatIdGeneration) \
  MAX_VERTICES 2                             \
  OUTPUT UNTOUCHED FEATURE_TYPE *           \
  OUTPUT CHOPPED FEATURE_TYPE *             \
# -----
# Now set up the transfer specifications to route the features
# to their final resting spot.
SHAPE roadGeometry                        \
  id %fid                                \
TABLE roadgeom                            \
  FEAT_ID %fid                            \
  X1      @Coordinate(X,0)                \
  Y1      @Coordinate(Y,0)                \
  X2      @Coordinate(X,1)                \
  Y2      @Coordinate(Y,1)

```

ClippingFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> ClippingFactory
[FACTORY_NAME <factory name>]
[INPUT CLIPPER FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]+
[INPUT CLIPPEE FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]+
[MULTICLIP (YES|NO|CLIPPERS_FIRST)]
[CLIPPER_ATTR_PREFIX <attribute prefix value>]
[CLIPPEE_ON_BOUNDARY (INSIDE|OUTSIDE|BOTH)]
[MERGE_CLIPPER_ATTRIBUTES [(YES|NO)]]
[DO_NOT_AGGREGATE [(YES|NO)]]
[GROUP_BY [<attribute name>]+]
[OUTPUT (CLIPPED_INSIDE|CLIPPED_OUTSIDE|
  INSIDE|OUTSIDE|EXTRA_CLIPPER|
  NONPOLY_CLIPPER)
  FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*

```

Overview

This factory is used to perform a geometric clipping operation on input features. It takes two types of features identified by the `CLIPPER` and `CLIPPEE` input tags.

Clipping features are identified by the tag `CLIPPER`. These features identify the area against which all `CLIPPEE` features are processed. The `CLIPPER` can be a polygon, donut, or aggregate feature. Any non-area clipping features that are encountered are output by the `NONPOLY_CLIPPER` tag. In the case of an aggregate `CLIPPER`, any non-polygonal parts it contains will be output as an aggregate by the `NONPOLY_CLIPPER` tag.

The “clippee” features are identified by the tag `CLIPPEE`. These are clipped in such a way that features totally within the `CLIPPER` feature are output from the factory via the `INSIDE` tagged output clause. Raster features that enter the factory via the `CLIPPEE` tag may only be clipped by a non-aggregate, rectangular feature. If a non-rectangular feature is used to clip a raster feature, the “clipper” will be readjusted to the bounding box of the non-rectangular feature.

The clipping operates in one of three ways according to the style of multiclippping selected by the tag `MULTICLIP`. If the default option of `NO` is selected, only one input clipper will be accepted. For each instance of the factory, the first polygonal feature that matches the `CLIPPER` specification is

used as the clipping polygon, and other polygonal features are output immediately by the `EXTRA_CLIPPER` tagged output clause.

Tip

Since `CLIPPEE` features must be held by the factory until the `CLIPPER` feature arrives, take care wherever possible to ensure that the `CLIPPER` feature arrives before any `CLIPPEE` features.

If the `YES` option for the `MULTICLIP` tag is selected, multiple `CLIPPER` features may be given to the clipping factory. The `CLIPPERS` will be applied to all the `CLIPPEES` in the order that they are input. In this mode, all the `CLIPPEE` features are held in a cache until all the `CLIPPERS` and `CLIPPEES` have arrived into the clipping factory. Then, when we are sure that all clippers have arrived, clipping begins.

The `CLIPPERS_FIRST` option for the `MULTICLIP` tag is the third and final mode which is a modification of the `YES` option that allows for better efficiency. If selected, the clipping factory will expect the first feature input to be `CLIPPERS`. Once the first `CLIPPEE` feature is given, all subsequent `CLIPPER` features will be output immediately by the `EXTRA_CLIPPER` tagged output clause. In this mode, no `CLIPPEE` features are held, and clipping begins when the first `CLIPPEE` arrives.

`CLIPPEE` features that intersect the `CLIPPER` features are broken up into pieces, with those pieces on the inside of the `CLIPPER` features being output by the `CLIPPED_INSIDE` tags. Those portions outside the clipping area are output via the `CLIPPED_OUTSIDE` tagged output clauses.

If a `CLIPPEE` feature is found to be inside a `CLIPPER`, or is clipped inside a `CLIPPER`, the inside parts are no longer clipped by other `CLIPPERS`. As such, results may vary depending on the order that the `CLIPPERS` are given to the clipping factory.

Features completely outside all of the `CLIPPER` features are output by the `OUTSIDE` tagged output clause. Features that are clipped outside of a `CLIPPER` will have their outside parts clipped by any subsequent `CLIPPERS`.

Raster features processed by the clipping factory are output as `CLIPPED_INSIDE` when the `CLIPPER` intersects some portion of the raster. Only the portion that falls inside the `CLIPPER` boundary is output in this fashion. If the `CLIPPER` specifies a boundary completely outside the raster feature, it is output via the `OUTSIDE` tag. This factory is currently unaffected by raster and/or palette sub-selection.

If `MERGE_CLIPPER_ATTRIBUTES` is specified, then whenever a clippee is found to be inside or clipped inside a `CLIPPER`, the attributes of the clipper will be copied to the inside parts of the output features. The `CLIPPER_ATTR_PREFIX` directive specifies a string prefix to be added to the attributes of a `CLIPPER` that

are copied to a `CLIPPEE`. If attributes of the same name already exist on the `CLIPPEE`, they will be overridden. If this prefix is not specified or is empty, the attributes will still be copied to the output features, but they will not overwrite any existing attributes of the same name.

The `CLIPPEE_ON_BOUNDARY` directive directs what action should be taken with clippee features that lie entirely on the boundary of the clipper. If the value `INSIDE` is given, then these features that lie on the boundary are output via the `INSIDE` tagged output clause. If the value `OUTSIDE` is given, then these features that lie on the boundary are output via the `OUTSIDE` tagged output clause. If `BOTH` is given, points and line segments on the boundary are duplicated and output as both `INSIDE` and `OUTSIDE`. If the `CLIPPEE_ON_BOUNDARY` directive is not specified, the default behavior is `INSIDE`.

This factory is most commonly used when an input data source covers an area much greater than the area of interest for the destination data set. This factory enables the FME to perform some rudimentary spatial selection operations, in addition to its attribution selection capabilities.

All of the output tags are optional.

The dimensionality of input features are maintained, with an exception noted in the following paragraph. This means that features resulting from input points are returned as points, features resulting from input lines are returned as lines, and features resulting from input polygons are returned as polygons. As well, features resulting from input aggregates are returned as aggregates, with their correct nesting depth (and relative order) preserved in the case of nested aggregates.

The factory may output aggregates and, in fact, aggregates may be output in cases where a non-aggregate vector feature is input. Imagine a windy river crossing into and out of the clipping region several times. The portion of the river inside the clipping region would be output as one aggregate of linear geometries and the portion outside the clipping region would be output as another aggregate of linear pieces representing the portion outside the clipping region. Specify the `DO_NOT_AGGREGATE` clause if you do not want aggregates generated where they did not previously exist. If the `DO_NOT_AGGREGATE` clause is used, then multiple pieces that result from a single `CLIPPEE` feature will be output as separate features with identical attributes.

Note Previously, elliptical arcs were stroked to lines and then clipped. Even if the arcs came out of the clipper as unclipped, they were still lines. Now elliptical arcs that are wholly inside or wholly outside all clippers will come out of the clipper as the original elliptical arcs, instead of arcs that have been stroked lines. Clipped elliptical arcs will still come out as stroked lines.

Note 3D lines are clipped 3D with the intersection interpolated along the line. Areas are clipped 3D with intersection points receiving elevation (z) from the CLIPPER.

If the `GROUP_BY` clause is given, then each `CLIPPEE` will only be clipped by `CLIPPERS` that contain the same values as itself in the attributes specified by this clause.

If the `FME_GEOMETRY_HANDLING` directive is set to yes in the mapping file, arcs and ellipses will be clipped without stroking. For example, the result of clipping an ellipse in half will result in an arc and a line connecting the two ends of the arc. Otherwise, all arcs and ellipses will be stroked prior to clipping.

Example

This example shows a collection of Shape files being filtered by a polygonal feature stored in a file called `boundary`.

```
# =====
# This factory is used to clip all features within the clip window
# defined by boundary and outputs the contained features and the
# portion of the features which are within the window. Features and
# portions of features which are outside the clipping window are
# discarded.
# The omission of the two output clauses dealing with outside
# features causes such features to be thrown away.
#
# The factory also uses the two error clauses to output any erroneous
# features which are encountered. The mapping file fragment shows
# how the data translation can be aborted when an erroneous
# condition occurs. If an erroneous condition occurs, the bad
# feature is logged and the translation is aborted via the
# @Abort() function
FACTORY_DEF SHAPE ClippingFactory
    INPUT CLIPPER          FEATURE_TYPE boundary
    INPUT CLIPPEE          FEATURE_TYPE lots
    INPUT CLIPPEE          FEATURE_TYPE roads
    INPUT CLIPPEE          FEATURE_TYPE water
    INPUT CLIPPEE          FEATURE_TYPE sewers
    MULTICLIP              NO
    OUTPUT CLIPPED_INSIDE  FEATURE_TYPE *
    OUTPUT INSIDE          FEATURE_TYPE *
    OUTPUT NONPOLY_CLIPPER FEATURE_TYPE BADCLIPPER
    @Abort(BadClipper)
    OUTPUT EXTRA_CLIPPER  FEATURE_TYPE ExtraClipper
    @Abort(ExtraClipper)
```


CloseFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> CloseFactory
[FACTORY_NAME <factory name>]
[INPUT (SHARED_BOUNDARY|INSIDE_POINT|BOUNDARY)
  FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]+
[GROUP_BY [<attribute name>]+]*
(END_NODDED|VERTEX_NODDED)
[OUTPUT (INSIDE_POINT|SHARED_BOUNDARY|
  PIP|LINE|POLYGON)
  FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*

```

Overview

The `CloseFactory` is very similar in operation to the `PolygonFactory`. However, it allows a single set of edges to be used with multiple groups of arcs to close polygons. The `PolygonFactory` does not allow such reusing of arc sets.

This factory is used when a noded mapsheet boundary is present only once in a file, but its edges must be used multiple times to close polygonal features. In such a case, two polygons are formed when the mapsheet boundary is used to close an otherwise unclosed polygon. In order to determine which of the two polygons is the desired one, an internal point is paired with the polygon. This feature is then output according to the `OUTPUT PIP` clause. The polygons not containing any point are output by the `OUTPUT POLYGON` clause.

This factory does not handle measures, and will convert complex geometries such as paths or text into lines when processing.

Assumptions

Within a group, there are no overlapping polygons. There must be one point for each polygon that is to be output. The shared boundaries must be noded correctly so that each group may use them to close polygons.

Input Tags

The `CloseFactory` uses the following input tags.

Tag	Description
SHARED_BOUNDARY	These are the boundary lines used to close polygons formed in any of the <i>groups</i> of BOUNDARY features.
INSIDE_POINT	These are the points that must be inside the closed polygons. They will be output with the polygons as point-in-polygon (PIP) geometry.
BOUNDARY	These are the boundary lines that will be grouped according to the GROUP_BY clause and formed into polygons, making use of the SHARED_BOUNDARY features. Resulting polygons that contain an INSIDE_POINT from the same group are returned as PIPs.

Output Tags

The `CloseFactory` supports the following output tags.

Tag	Description
SHARED_BOUNDARY	The original shared boundaries come out unchanged. If this is not specified, then such features are discarded.
LINE	After attempting to form polygons with the set of boundary and shared boundary elements, anything that did not close and was left as a line is output via this clause. It will have the GROUP_BY attributes added to it. If this is not specified, then such features will be discarded.
PIP	The formed polygons that contain a point come out here. They have point-in-polygon geometry, and all attributes of the point are merged with them. If this is not specified, then such features will be discarded.
INSIDE_POINT	Any points that came in but were never placed inside a polygon are output with this clause. If a point is placed in a polygon, it is <i>not</i> output with this clause. If this is not specified, such unused points are discarded.
POLYGON	Normally this is not specified. However, if it is then any polygons formed that <i>did not</i> have a point inside of them are output with this clause. It will have the GROUP_BY attributes added to it.

Example

The following example illustrates the use of `CloseFactory` to form polygons from IGDS lines and points grouped by graphic group. The shared edges may be used by all groups to close polygons.

```

FACTORY_DEF IGDS CloseFactory \
  INPUT SHARED_BOUNDARY FEATURE_TYPE 62 igds_type igds_line \
  INPUT BOUNDARY FEATURE_TYPE 26 igds_type igds_line \
  INPUT INSIDE_POINT FEATURE_TYPE 26 igds_type igds_text_node \
  END_NODDED \
  GROUP_BY igds_graphic_group \
  OUTPUT PIP FEATURE_TYPE pips \
  OUTPUT INSIDE_POINT FEATURE_TYPE rejpt \
  OUTPUT POLYGON FEATURE_TYPE poly \
  OUTPUT LINE FEATURE_TYPE rejects \
  OUTPUT SHARED_BOUNDARY FEATURE_TYPE shared

```

CommonSegmentFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> CommonSegmentFactory
  [FACTORY_NAME <factory name>]
  [INPUT (BASE|CANDIDATE) FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [GROUP_BY [<attribute name>+]*]
  [OUTPUT (OVERLAP|NO_OVERLAP|BASE)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory tests to see which of the features entering by the *CANDIDATE* input clause have even one segment in common with any feature entering by way of the *BASE* input clause. If a *CANDIDATE* feature *does* share one segment with some *BASE* feature, then it is output with the *OUTPUT OVERLAP* clause. If the candidate does *not* share any segment, then it is output via the *OUTPUT NO_OVERLAP* clause. As the factory shuts down, all *BASE* features are output unmodified through the *BASE* output clause.

The *GROUP_BY* clause allows the candidates and base features to be clustered on the basis of some set of attribute values.

All attributes present on any input feature when it enters the factory, will be present when the feature leaves it. The factory makes no changes to any features that flow through it.

Whenever possible the *BASE* feature set should be kept as small as possible to reduce the memory requirements of this factory.

Assumptions

The only overlap that can be detected is one where at least one segment connects *precisely* the same two vertices in both the *BASE* and *CANDIDATE* features.

Input Tags

The `CommonSegmentFactory` supports the following input tags.

Tag	Description
BASE	The set of features to be searched for a common segment by each candidate feature.
CANDIDATE	The features to be categorized as either having a common segment with the base set or not.

Output Tags

The `CommonSegmentFactory` supports the following output tags.

Tag	Description
OVERLAP	Any candidate feature that shares a common segment with some feature in the base set.
NO_OVERLAP	Any candidate feature that does <i>not</i> share a common segment with some feature in the base set.
BASE	The original, unchanged input base features.

Example

The following example categorizes all `road` features as either having a segment that overlaps some segment in a `boundary` feature or not. If a `road` feature does overlap, it is given an attribute `overlaps` with the value `yes`, otherwise this attribute is given the value `no`. In any case, the `boundary` features are output unchanged.

```
FACTORY_DEF SHAPE_IN CommonSegmentFactory      \  
  INPUT BASE FEATURE_TYPE boundary              \  
  INPUT CANDIDATE FEATURE_TYPE road             \  
  OUTPUT OVERLAP FEATURE_TYPE road overlaps yes \  
  OUTPUT NO_OVERLAP FEATURE_TYPE road overlaps no \  
  OUTPUT BASE FEATURE_TYPE *
```

ConnectionFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> ConnectionFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute Value>]*
    [<feature function>]*]*
  [BREAK_BEFORE_FIELD_CHANGE [<attribute name>]+]
  [BREAK_AFTER_FIELD_CHANGE [<attribute name>]+]
  [FEATURE_CONNECTION (ORDERED|NODED)]
  [END_CURRENT_WHEN <value> <operator> <value>]+
  [START_NEW_WHEN <value> <operator> <value>]+
  [LIST_NAME <list name>{}]
  [CLOSE_WHEN <value> <operator> <value>]+
  [OUTPUT (POINT|LINE|POLYGON|BAD_INPUT|PATH)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory is used to construct new features from a collection of input features. The factory accepts features with point or linear geometry and outputs features that have a combination of the input features' geometry. Any feature received that does not have a geometry of POINT or LINE is output by the BAD_INPUT output tag.

At any given time the factory is creating only one feature. The current feature being constructed is output whenever any of the following cases is true:

- a. The values of any of the **BREAK_BEFORE_FIELD_CHANGE** fields change from one feature to the next. When this occurs, the newly received input feature is not part of the flushed feature but rather is part of the next feature to be constructed. Values of attributes named in these clauses are transferred to the output features.
- b. The values of any of the **BREAK_AFTER_FIELD_CHANGE** fields change from one feature to the next. When this occurs, the newly received input feature is part of the flushed feature. Values of attributes named in these clauses are transferred to the output features.
- c. The values of any of the **END_CURRENT_WHEN** clauses evaluate to true. When this occurs, the newly received input feature is part of the flushed feature.
- d. The values of any of the **START_NEW_WHEN** clauses evaluate to true. When this occurs, the newly received input feature is not part of the flushed feature but instead is the first part of the next feature.

The clause specified by **CLOSE_WHEN** is only evaluated when one of the clauses described in a through d are true. If the **CLOSE_WHEN** clause is true, then the

output feature is converted into a polygon provided that the feature has at least 3 coordinates. Closed features are output via the `POLYGON` output tag.

When the factory receives a point feature, it simply adds the feature to the end of the current feature being constructed, unless one of the break conditions described above is true.

When the factory receives a linear feature, it does the following:

- a. If `LINEAR_FEATURE_CONNECTION` is `ORDERED` then the factory merely appends the coordinates to the end of the feature being constructed.
- b. If `LINEAR_FEATURE_CONNECTION` is `NODED`, the factory adds the coordinates to the end but removes the first coordinate of subsequent features as they are duplicated from one component feature to the next.

The attributes placed on output features are a concatenation of the first input component feature and all subsequent component features with no geometry. Additional attributes may be placed on the output features using the `LIST_NAME` clause.

The optional `LIST_NAME` clause is used to retain and associate attributes that came from each component of the output feature. When the output feature is created, all attributes of each feature joining to form the output feature are added as members of the specified attribute list. The index in the list corresponds to the index of the feature's contribution to the output feature's final geometry.

If the `FME_GEOMETRY_HANDLING` directive is set to `yes` in the mapping file, arcs will be allowed as input; they will otherwise be ignored.

Path Assembly Mode

If the `PATH` output tag is given, the factory operates in a different mode. It accepts linear (i.e. line, arc, or path) features and assembles paths from them.

Paths are assembled in the same order as the segments enter the factory. All the same break methods described above can be used to designate when to output the path currently under construction.

Input paths are allowed. Note that if any paths are received and `LIST_NAME` is given, then the number of list elements will differ from the number of path segments.

The `CLOSE_WHEN` and `FEATURE_CONNECTION` clauses are invalid in this mode. In addition, the `POLYGON`, `LINE`, and `POINT` output tags may not be used.

The output features in this mode are always paths (that is, even if only one segment arrives, it will be output as the sole segment of a path). Closed paths will be output as paths, not changed to polygons.

Assumptions

The features to be connected are assumed to enter the factory in the correct order. If features are to be connected by joining at equivalent start and end points, then the `ArcFactory` should be used.

Output Tags

The `ConnectionFactory` supports the following output tags.

Tag	Description
POLYGON	Polygon features constructed by the factory are output by this tag.
LINE	Linear features constructed by the factory are output using this tag.
POINT	Features which remain as points are output here.
PATH	Specifying this output tag causes the factory to operate in Path Assembly Mode; please see the description of this mode above.
BAD_INPUT	Any feature whose geometry is not a point or a line (in regular mode) or an arc/line/path (in path assembly mode) is output with this tag.

The following example constructs linear features from point features. The input file has the following definition:

The values of `ID`, `X`, and `Y` require no explanation. The values for `NODTYPE` are as follows:

This factory definition connects all nodes into linear and polygonal features:

```

FACTORY_DEF TABLE ConnectionFactory
INPUT FEATURE_TYPE *
END_CURRENT_WHEN &NODTYPE = 4
END_CURRENT_WHEN &NODTYPE = 3
CLOSE_WHEN &NODTYPE = 4
OUTPUT LINE FEATURE_TYPE line
OUTPUT POLYGON FEATURE_TYPE polygon

```

ConvexHullFactory

Syntax

```
FACTORY_DEF <ReaderKeyword> ConvexHullFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [GROUP_BY [<attribute name>+]*]
  [OUTPUT (ORIGINAL|CONVEX_HULL)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
```

Overview

This factory takes a set of point, linear, polygonal, and/or aggregate features and computes its convex hull. The convex hull is defined as the minimum enclosing convex polygon for all input features. A convex polygon is one where no interior angle is greater than 180 degrees. In lay terms, the effect is similar to tightening a rubber band around the features.

The input features are partitioned into groups based on attribute values by way of the `GROUP_BY` clause, and one convex hull feature is output for each group. If the `GROUP_BY` clause is not specified, then all input features will be processed together and only a single convex hull will be output.

No attributes are carried across from the `INPUT` features to the `OUTPUT CONVEX_HULL` features. However, all `OUTPUT CONVEX_HULL` features are assigned the `GROUP_BY` attributes before being output.

The `OUTPUT ORIGINAL` features are untouched as a result of their visit to the factory. If this clause is not specified, then the original features will be deleted.

This factory differs from the `@ConvexHull()` function in that the function computes the convex hull of a single feature, whereas this factory computes the convex hull of a group of features.

If the `FME_GEOMETRY_HANDLING` directive is set to yes in the mapping file, all arcs, ellipses, and text will be stroked prior to the computation of the convex hull; otherwise, the center point of these geometries will be used during the computation of the convex hull.

The algorithm used to compute the convex hull is used by permission of the notice below:

Ken Clarkson wrote this [convex hull algorithm]. Copyright (©) 1996 by AT&T. Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

THIS SOFTWARE IS BEING PROVIDED “AS IS”, WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHORS NOR AT&T MAKE ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

Assumptions

None.

Output Tags

The ConvexHullFactory supports the following output tags.

Tag	Description
CONVEX_HULL	The convex hull of the group of features, stored as a simple polygon.
ORIGINAL	The original, unchanged input features.

Example

The following example takes all the points from a Shape file named `shotpoints` and groups them by their `CLASS` attribute. A convex hull is output for each group. The original points are not output by the factory.

```
SHAPE_IN_DEF shotpoints \
  SHAPE_GEOMETRY shape_point \
  CLASS char(10)
SHAPE_OUT_DEF hulls \
  SHAPE_GEOMETRY shape_polygon \
  CLASS char(10)
FACTORY_DEF SHAPE_IN ConvexHullFactory \
  INPUT FEATURE_TYPE shotpoints \
  GROUP_BY CLASS \
  OUTPUT CONVEX_HULL FEATURE_TYPE created_hulls \
SHAPE_IN created_hulls \
  CLASS %class
SHAPE_OUT hulls \
  CLASS %class
```

CorrelationFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> CorrelationFactory
  [FACTORY_NAME <factory name>]
  [INPUT_FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function> ]]*
TABLE_DEF (ASCII|CAT|SCAT|CSV|DBF|DATABASE)
  [<attrname> <attrtype>]+
TABLE_LOCATION <filename>
  [TABLE_ID <tableId>]
  [CORRELATION_SPEC <column name> <value>]+
  [OUTPUT_FEATURE_TYPE_COLUMN <column name>]
  [CORRELATION_COLUMNS <input attrname column name>
    <output attrname column name>
    [<default attribute value column name>]]+
  [OUTPUT (CORRELATED|NOT_CORRELATED)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory is used to perform correlation operations on features. It takes features and manipulates the attributes based on the directions stored in an associated database table. The table contains instructions on which attributes should be added and possibly which features should be cloned. Effective use of this factory eliminates the need for complex correlation lines in the mapping file. Combinations of the following operations can be performed on features through this factory:

Operation	Description
Duplicate attribute	Specifying input and output attribute names in the columns defined on the <code>CORRELATION_COLUMNS</code> clause duplicates the attribute present on a feature. A default value may be given if the attribute is not present. The result is that the output attribute name will be added to the feature with the value the named input attribute has.
Add attribute	Specifying an output but no input attribute name in the columns defined on the <code>CORRELATION_COLUMNS</code> clause adds the new attribute to a feature. The value may be specified by defining a default.
Remove attribute	Specifying an input but no output attribute name in the columns defined on the <code>CORRELATION_COLUMNS</code> clause removes the attribute from the feature.

More Information

The first keyword on the `TABLE_DEF` clause defines the table type. If the table type is `DATABASE`, then the `TABLE_LOCATION` is interpreted as the database ID – either an ODBC data source or an Oracle instance. For file-based types, the `TABLE_LOCATION` is the full path name of the file.

More Information

If you want specifics on the field types and special fields for each table type, see the discussion under the heading *DEF* in the *FME Readers and Writers* manual in the chapter titled *Relational Table Reader/Writer*.

Keywords <column name> identify table columns.

The `CORRELATION_SPEC` clause determines which table rows match each input feature. Once matches are made, each table row dictates the actions that will be performed on the feature. A `CORRELATION_SPEC` is considered to match a row to a feature if the value stored in the `<column name>` column of the row matches the value of `<value>`. The value of `<value>` may be either a literal constant value, for example `river`; an attribute name preceded by the value-of operator, for example `&color`; or an attribute function, for example `@FeatureType()`. If

it's an attribute value function, the function will be executed on the current feature and the result will be used for the matching row.

If no matching rows are found, the feature will be output unchanged by way of the `OUTPUT_NOT_CORRELATED` clause.



In a future version of the FME, if no `CORRELATION_SPEC` clauses are defined, every input feature will be considered to match all rows in the table.

Features that find matching rows in the table are output with the new feature type, as specified in the `OUTPUT_FEATURE_TYPE_COLUMN`. If more than one feature type is specified in this column in different matching rows, more than one copy of the input feature will be produced. If the `OUTPUT_FEATURE_TYPE_COLUMN` clause is omitted, then the feature type of all output features will be the same as they were when input.

Before the feature is output, the `CORRELATION_COLUMNS` clause settings are used to modify the attributes of the feature. For each of these clauses, the value in the `<input attrname column name>` specifies the name of an attribute that will be duplicated as a new attribute with the name found in the `<output attrname column name>`. This attribute modification, defined on the `CORRELATION_COLUMNS` clauses, is done for each row that matches the feature.

If several copies of the input feature are being output because more than one output feature type is specified, the new attributes will only be added to the output feature with the feature type specified in the `OUTPUT_FEATURE_TYPE_COLUMN` of that particular row.

If the input feature does not have any value associated with the input attribute and the `<default attribute value column name>` is specified, the value in this column will be supplied to the output attribute on the output feature.

If the `<input attrname column name>` column is blank, then the new attribute is simply added with the default value.

If the `<output attrname column name>` column is blank, the original attribute will be removed.

Assumptions

None.

Output Tags

The `CorrelationFactory` supports the following output tags.

Tag	Description
CORRELATED	These are the features that result from applying the manipulations specified in the associated database table.
NOT_CORRELATED	These are the original, unchanged input features that had no matching entries in the associated database table.

Example 1

The following example alters all text and symbol features as specified in the csv file named `example1.csv`. The factory definition is shown below:

```

FACTORY_DEF IGDS CorrelationFactory
  INPUT FEATURE_TYPE *
  TABLE_DEF CSV
    CSV_SKIP_LINES 2
    inputFeatureType char
    outputFeatureType char
    oldAttribute char
    newAttribute char
    defaultValue char
  TABLE_LOCATION "$(FME_CF_DIR)/example1.csv"
  CORRELATION_SPEC inputFeatureType @FeatureType()
  OUTPUT_FEATURE_TYPE_COLUMN outputFeatureType
  CORRELATION_COLUMNS oldAttribute newAttribute defaultValue
  OUTPUT NOT_CORRELATED FEATURE_TYPE *
  OUTPUT CORRELATED FEATURE_TYPE *

```

The contents of `example1.csv` is:

```
inType,outType,old,new,defaultValue
-----
symbol,phone booth,angle,rotation,360.0
text,label,name,text_string,missing label
```

With the definitions above, all `symbol` features will have their feature types changed to `phone booth` and a new attribute named `rotation` will be added which will have the same value as the existing attribute `angle`. If no attribute called `angle` exists, the value of `360.0` will be given.

All text features will have their feature types changed to `label` and a new attribute called `text_string` with the same value as the existing attribute `name` will be added. If no attribute called `name` exists, the value `missing` label will be given.

For example, if the following two features entered the factory:

```
1) Feature Type: text
   attribute name: name attribute value: highway
2) Feature Type: symbol
   attribute name: angle attribute value: 45
```

the following two features will leave the factory from the OUTPUT_CORRELATED clause:

```
1) Feature Type: label
   attribute name: name attribute value: highway
   attribute name: text_string attribute value: highway
2) Feature Type: phone booth
   attribute name: angle attribute value: 45
   attribute name: rotation attribute value: 45
```

Example 2

The following example alters all text and symbol features as specified in the CAT file named `example2.cat`. The factory definition is:

```
FACTORY_DEF SHAPE CorrelationFactory \
INPUT FEATURE_TYPE * \
TABLE_DEF CAT \
    inputFeatureType String(6,1) \
    outputFeatureType String(11,8) \
    oldAttr1 String(5,20) \
    newAttr1 String(11,26) \
    defaultValue1 String(13,38) \
    oldAttr2 String(5,52) \
    newAttr2 String(8,58) \
    defaultValue2 String(7,67) \
TABLE_LOCATION "$(FME_CF_DIR)/example2.cat" \
CORRELATION_SPEC inputFeatureType @FeatureType() \
OUTPUT_FEATURE_TYPE_COLUMN outputFeatureType \
CORRELATION_COLUMNS oldAttr1 newAttr1 defaultValue1 \
CORRELATION_COLUMNS oldAttr2 newAttr2 defaultValue2 \
OUTPUT NOT_CORRELATED FEATURE_TYPE * \
OUTPUT CORRELATED FEATURE_TYPE *
```

The contents of `example2.cat` (a column-aligned-text file) are:

```
symbol phone booth angle rotation    360.0          color style    black
symbol phone booth                orientation center
text                               name text_string missing label type surface unknown
text  label                       name                               width wideness thick
```


With the definitions above, all `symbol` features will have their feature types changed to `phone booth` and a new attribute `rotation` will be added with the same value as the existing attribute `angle`. If no attribute `angle` exists, the value of `360.0` will be given. As well, a new attribute `style` will be added with the same value as the existing attribute `color`. If no attribute `color` exists, the value of `black` will be given. Finally, all `symbol` features—now with the feature type `phone booth`—will have a new attribute `orientation` with the value of `center` added.

All `text` features will be cloned or divided into two output features: one copy will keep the feature type `text`, while the other copy will have the feature type `label`.

The copy that keeps the feature type `text` will have a new attribute `text_string` added with the same value as the existing attribute `name`. If no attribute `name` exists, the value `missing label` will be given. As well, a new attribute `surface` will be added having the same value as the existing attribute `type`. If no attribute `type` exists, the value `unknown` will be given.

The copy of the `text` feature that has its feature type changed to `label` will have the attribute `name` deleted. Also, a new attribute `wideness` will be added with the same value as the existing attribute `width`. If no attribute `width` exists, the value `thick` will be given.

For example, if the following two features entered the factory:

```
1)Feature Type: text
attribute name: name attribute value: highway
attribute name: type attribute value: paved
2)Feature Type: symbol
attribute name: angle attribute value: 45
attribute name: color attribute value: red
```

the following three features will leave the factory from the `OUTPUT_CORRELATED` clause:

```
1)Feature Type: text
attribute name: name attribute value: highway
attribute name: text_string attribute value: highway
attribute name: type attribute value: paved
attribute name: surface attribute value: paved
2)Feature Type: label
attribute name: type attribute value: paved
attribute name: wideness attribute value: thick
3)Feature Type: phone booth
attribute name: angle attribute value: 45
attribute name: rotation attribute value: 45
attribute name: color attribute value: red
attribute name: style attribute value: red
attribute name: orientation attribute value: center
```

CreationFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> CreationFactory
[FACTORY_NAME <factory name>]
[( 2D_GEOMETRY <x0> <y0> <x1> <y1> ... <xN> <yN> |
  3D_GEOMETRY <x0> <y0> <z0>
    <x1> <y1> <z1> ... <xN> <yN> <z1>)]
[COORDINATE_SYSTEM <coordSysId>]
[NUMBER_TO_CREATE <value>]
[CREATE_AT_END [yes|no]]
[OUTPUT FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*

```

Overview

This factory creates new features. By default, the features it creates are guaranteed to come before any other features. If the `CREATE_AT_END` clause is specified, then the features created come after all other features have passed the factory.

Even if the FME reader produces **no** features, the factory will create its features and pass them along.

If the coordinates form a point, line, polygon, or donut polygon, the feature(s) created will be tagged as such. If a point feature is created and the attributes `fme_primary_axis` and `fme_secondary_axis` are added using the `OUTPUT` clause, then the feature will be tagged as an ellipse (i.e., its `fme_type` will be `fme_ellipse`). However, if a point feature is created and the attributes `fme_primary_axis`, `fme_secondary_axis`, `fme_start_angle`, and `fme_sweep_angle` are added using the `OUTPUT` clause, then the feature will be tagged as an arc (i.e., its `fme_type` will be `fme_arc`). If a point feature is created and the attributes `fme_text_string` and `fme_text_size` are supplied using the `OUTPUT` clause, then the feature will be tagged as a text feature (i.e., its `fme_type` will be `fme_text`).

Only one of `2D_GEOMETRY` or `3D_GEOMETRY` may be specified. If neither one is specified, then features with no geometry are created and their `fme_type` will be set to `fme_no_geom`.

One feature is output for each `OUTPUT` clause. Attributes and feature types may be supplied on the `OUTPUT` clause in the usual way. If `NUMBER_TO_CREATE` is specified, then the number of features specified will be output on each output clause. In addition, each created feature will have an attribute called `creation_instance` added to it which holds the creation sequence number (starting with 0) of the feature. The `NUMBER_TO_CREATE` may be specified as an

integer, or as an FME function call which will return the number of features to create.

If no `COORDINATE_SYSTEM` is specified, then the feature(s) will be created to have the same coordinate system as the first feature to arrive at the `CreationFactory`.

Assumptions

None.

Output Tags

The `CreationFactory` does not use output tags.

Example

The following examples show how the `CreationFactory` can be used to inject features into the processing stream. The last example creates a feature that is a donut polygon.

```

FACTORY_DEF * CreationFactory           \
COORDINATE_SYSTEM UTM10-83              \
2D_GEOMETRY 3 4 4 5                     \
OUTPUT FEATURE_TYPE one @Log()           \
OUTPUT FEATURE_TYPE two @Log()

FACTORY_DEF * CreationFactory           \
COORDINATE_SYSTEM UTM10-83              \
2D_GEOMETRY 3 4                         \
OUTPUT FEATURE_TYPE point @Log()         \
OUTPUT FEATURE_TYPE point @Log()

FACTORY_DEF * CreationFactory           \
COORDINATE_SYSTEM UTM10-83              \
NUMBER_TO_CREATE @Evaluate("3 * 4")      \
2D_GEOMETRY 3 4                         \
OUTPUT FEATURE_TYPE point @Log()

FACTORY_DEF * CreationFactory           \
3D_GEOMETRY 3 4 2                       \
OUTPUT FEATURE_TYPE 3dpoint @Log()

FACTORY_DEF * CreationFactory           \
COORDINATE_SYSTEM UTM10-83              \
OUTPUT FEATURE_TYPE nothing @Log()

FACTORY_DEF * CreationFactory           \
2D_GEOMETRY 0 0                         \
                1 0                     \
                1 1                     \
                1 1                     \

```

```
0 1
0 0
0.2 0.2
0.8 0.2
0.8 0.8
0.2 0.8
0.2 0.2
OUTPUT FEATURE_TYPE poly @Log()
```

/
/
/
/
/
/
/

CSGFactory

Syntax

```
FACTORY_DEF <ReaderKeyword> CSGFactory
  [FACTORY_NAME <factory name>]
  [INPUT A|B FEATURE_TYPE <factory type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [GROUP_BY [<attribute name>]+]
  [OUTPUT UNION|DIFFERENCE|INTERSECTION|UNUSED *
FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
```

Overview

The CSGFactory performs Union, Difference and Intersection Boolean operations on a solid geometry feature from input A against another solid geometry feature from input B, and outputs the result through the output tags UNION, DIFFERENCE and INTERSECTION respectively. The output is a feature with a CSG solid geometry.

The factory only takes one solid feature from each input A and B. Any extra features or non-solid features are output through the `UNUSED` tag. It does not output anything through `UNION`, `DIFFERENCE` and `INTERSECTION` tags unless it can find one solid feature from each input tag.

The `GROUP_BY` clause can be used to perform a Boolean operation for a pair of solids (one from each input tag) that has the same attributes specified by this clause.

Output Tags

The CSGFactory supports the following output tags.

Tag	Description
UNION	Result of solid A union with solid B
DIFFERENCE	Result of solid A minus solid B
INTERSECTION	Result of solid A intersects with solid b
UNUSED	Features that are rejected by the factory

Examples

```
FACTORY_DEF * CSGFactory
INPUT A FEATURE TYPE *
```

```
INPUT B FEATURE_TYPE *
[GROUP_BY [<field name>+]*]
[OUTPUT UNUSED FEATURE_TYPE *]
[OUTPUT UNION FEATURE_TYPE *]
[OUTPUT DIFFERENCE FEATURE_TYPE *]
[OUTPUT INTERSECTION FEATURE_TYPE *]
```

DeaggregateFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> DeaggregateFactory
[FACTORY_NAME <factory name>]
[INPUT FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*
[LIST_NAME <list name>{}]
[PART_NUMBER_FIELD <attribute name>]
[RECURSIVE [(yes|no)]]
[SET_FME_TYPE [(yes|no)]]
[PATH_SPLIT_MODE [(yes|no)]]
[OUTPUT (DONUT|LINE|PIP|POINT|POLYGON|AGGREGATE|
  SEGMENT|NON_PATH)
  FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*

```

Overview

This factory takes a feature with aggregate geometry and breaks it into several features, one for each geometry fragment it contained. Each piece gets a complete copy of the original aggregate feature's attributes.

The optional `LIST_NAME` clause is used to associate attributes with each member of the aggregate. When the aggregate is broken up, any elements of the attribute list corresponding to the geometry will be added to the feature that is output in the same manner as with the `ElementFactory`.

If the optional `PART_NUMBER_FIELD` clause is specified, then each deaggregated output feature is given an attribute containing the part's (deaggregated feature) index within the original aggregate feature. This clause will not apply if `RECURSIVE` clause is also used.

The optional `RECURSIVE` keyword is used to deaggregate nested aggregates recursively. The final output will contain no aggregates, only the underlying geometry components. If the keyword is not used, the default action is no recursion and any nested aggregates will not be broken apart.

If `SET_FME_TYPE` is given, all features output will have an appropriate `fme_type` attribute set (which will be the same as the input feature's type, if it exists).

`PATH_SPLIT_MODE` designates that the factory is to split paths that are input and output their component segments, *instead* of splitting aggregate geometries.

If an aggregate had a member for a geometry type not represented in any `OUTPUT` clause, that portion of the geometry will be discarded.

Tip

The `DeaggregateFactory` is similar to the `ElementFactory`. The `ElementFactory` breaks attribute lists into features, whereas the `DeaggregateFactory` breaks aggregated geometry into features.

Assumptions

Features entering the `DeaggregateFactory` should have aggregate geometry. Aggregate geometry may be found on features coming from several formats, or from features created by the `AggregateFactory`. If a feature that is not an aggregate enters the factory, then that feature is output via the appropriate `OUTPUT` clause, according to the geometry it carries.

The `DeaggregateFactory` supports the following output tags.

The feature below matches the input specification of this factory definition and enters the factory.

	Feature Type: ContourAggregate	
	Attribute Name	Value
	position.geometry.qualifier	definite
	position.geometry.value	20
	Coordinates: Line 1: (477553,5360181,20) (477554,5360182,20) Line 2: (377553,4360181,20) (377554,4360182,20)	

The following two features are output from the factory.

	Feature Type: Contour::TRIM	
	Attribute Name	Value
	position.geometry.qualifier	definite
	position.geometry.value	20
	featNum	0
Coordinates: (477553,5360181,20) (477554,5360182,20)		

	Feature Type: Contour::TRIM	
	Attribute Name	Value
	position.geometry.qualifier	definite
	position.geometry.value	20
	featNum	1
Coordinates: (377553,4360181,20) (377554,4360182,20)		

Note the featNum attribute that was added as each of the features left the factory.

DEMDistanceFactory

Note: This factory has been renamed from **NVectors1RasterToNRastersFactory**. It is not supported by FME Base Edition.

Syntax

```
FACTORY_DEF <ReaderKeyword>
    DEMDistanceFactory
[FACTORY_NAME <factory name>]
[INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]* ]*
ACTION <processing type>
[GROUP_BY [<field name>+]*]
[OUTPUT RASTER
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
```

Overview

This factory was designed to be used in situations where processing is done with a single raster and many vectors. The processing pattern is defined as follows: for each group, incoming valid vector features are stored until a reference raster is received for this group. Immediately following the reception of this raster, any stored vectors are processed using the raster. The vectors that come after reception of the raster are processed as they are received.

The processing that will take place depends on the value specified for the **ACTION** parameter. However, every scenario will involve outputting one raster feature for every vector feature that the factory receives. A single different raster may be used for every individual group of incoming features. The **GROUP_BY** parameter should be used for the purpose of defining such feature groups.

Currently, the factory supports only the following **ACTION**:

DEM_DIST

In this mode, the factory requires three-dimensional line segments as input vectors, and numeric Digital Elevation Model (DEM) formats as input rasters. For each 3D line segment, a numeric DEM raster is output. The data contained in each cell of each output DEM is the shortest 3D distance from the corresponding cell of the input DEM to the line segment being considered.

Assumptions

For the **DEM_DIST** processing type, incoming line features must be three-dimensional. The incoming raster feature must contain valid elevation data. The

factory will produce output data that will only be accessible using numeric, continuous data calls.

Output Tag

Tag	Description
RASTER	The raster created for each vector feature will be output via the port associated with this tag. Note that the output features will contain any attributes that the corresponding input vector features had, with the exception of the <code>fme_type</code> and <code>fme_geometry</code> attributes (in <code>dem_dist</code> processing mode). Attributes contained on the reference input raster will not be present on output features.

DonutFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> DonutFactory
[FACTORY_NAME <factory name>]
[INPUT FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*
[DROP_HOLES [(yes|no)]]
[TAG_HOLES [(yes|no)]]
[BREAK_BEFORE_FIELD_CHANGE [<attribute name>]+]
[BREAK_AFTER_FIELD_CHANGE [<attribute name>]+]
[FLUSH_BEFORE_CURRENT_WHEN <value> <operator> <value>]
[FLUSH_AFTER_CURRENT_WHEN <value> <operator> <value>]
[FME_GEOMETRY_HANDLING [(yes|no)]]
[ALLOW_CYCLES [(yes|no)]]
[SPLIT_INPUT_DONUTS [(yes|no)]]
[HOLE_LIST <list attribute name>{}]
[GROUP_BY [<attribute name>+]*]
[OUTPUT (DONUT|LINE|PIP|POINT|POLYGON)
  FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*

```

Overview

This factory takes a series of features and joins them together based on spatial enclosure. Only features with polygon, donut polygon, or point geometry are processed by the factory. If a linear feature is found, then it is output according to the **OUTPUT LINE** clause. Once all input features have been collected, the factory produces an internal spatial ordering of all components.

If the **FME_GEOMETRY_HANDLING** directive is set to “yes” in the mapping file, ellipses will be handled as input polygons; they will otherwise be ignored.

The factory assumes that all input is topologically clean and that areas do not overlap.

If the **DROP_HOLES** directive is set to *yes*, any polygons enclosed within another polygon will be deleted. Holes of holes are kept. If **DROP_HOLES** is set to *no*, then all polygons will be output. **DROP_HOLES** drops the hole polygon, leaving a hole-shaped void in place of the hole polygon. If **DROP_HOLES** is not specified, *yes* is assumed.

If the **TAG_HOLES** directive is set to *yes*, any polygons enclosed within another polygon will have the attribute *tagged_hole* added to them with a value of *yes*. If the **TAG_HOLES** directive is set to *no*, any features not enclosed will have a value of *no* in the *tagged_hole* attribute when they exit. If **TAG_HOLES** is not specified, *yes* is assumed.

`ALLOW_CYCLES` specifies that coordinate “cycles” within a polygon are allowable; such polygons might be considered invalid by other parts of FME or by output formats. A “cycle” is a line segment that occurs twice in the same polygon's boundary (once in each direction).

If `SPLIT_INPUT_DONUTS` is specified, then donuts that are input to the factory will be split into their component rings before the donut-building algorithm is executed. If this directive is absent, then the behavior is undefined when there exist input donut features; in most cases such features will be output unmodified.

If `HOLE_LIST` is given, then a list will be created on each output feature, containing an element for each input feature which became a hole on that geometry, in order of appearance.

If the `FLUSH_BEFORE_CURRENT_WHEN` directive is specified, the factory performs the test defined by this clause every time it receives an input feature. If the result of the test is true, then the factory flushes out all stored features via the `OUTPUT` clauses before processing the input feature. If the result is false, the feature is processed and the factory is not flushed. The `FLUSH_AFTER_CURRENT_WHEN` tag operates identically except the factory is flushed after the current input feature processed instead of before.

The `<operator>` is one of `<`, `>`, `=`, `!=`, `>=`, `<=`.

The `<value>` may be a literal constant, an attribute name preceded by the value-of operator (`&`), or an attribute value function. If it is an attribute value function, the function is executed on the current feature and the result will be used for the test.

The example `FLUSH_BEFORE_CURRENT_WHEN` and `FLUSH_AFTER_CURRENT_WHEN` clauses include:

```
FLUSH_BEFORE_CURRENT_WHEN @Area() < 100
FLUSH_AFTER_CURRENT_WHEN &numLanes > 2
FLUSH_BEFORE_CURRENT_WHEN "Joe" = "Jerry"
```

At run-time, the `DonutFactory` decides to invoke numeric comparisons or string comparisons, as greater than and less than, that have different meanings depending on the type of the operands. If both arguments may be converted to numbers, then numeric comparisons are used, otherwise string comparisons are used.

The factory performs the following steps each time it is flushed, or once the last feature is input to the factory:

- 1 First, the factory determines the nesting of all input polygons.
- 2 At the conclusion of this operation, all polygons and donut polygons have been identified.

- The current set of features is also output whenever any of the following cases is true:

- Additional statistical information about the operation of the DonutFactory may be output by specifying `FME_DEBUG DonutFactory` in the mapping file. Search *Mapping File Debugging* in *FME Universal Translator on-line help*.

The `DonutFactory` assumes that the model is topologically clean and that no polygons within a group intersect. It is further assumed that each point is only in one polygon, and that each polygon has only one point.

The `DonutFactory` supports the following output tags.

Tag	Description
DONUT	All donut polygons that were successfully constructed but contained no points.

LINE	All line objects sent into the model. The geometry of these features is unchanged.
PIP	All Point-In-Polygon (PIP) features. The geometry of these features consists of a polygon or donut polygon and an inner point. The point's attributes are merged with the polygon's.
POINT	All point features that weren't found to be in any polygon.
POLYGON	All polygon features that don't contain any holes or an inside point.

Example

The following example defines a `DonutFactory` that accepts two types of features. The input polygons are of the type `ForestCoverPolygons`, and are nested by the `DonutFactory`. The `InteriorPoint` features are point features that the `DonutFactory` attempts to match up to polygon features in which they are contained.

The example outputs the following feature types:

- `POLYGON` – These are polygonal features for which no internal polygon or internal point was input into the factory.
- `DONUT` – These are polygonal features for which internal polygons were found, but no internal points were found.
- `POINT` – These are the point features for which no enclosing polygon was found.
- `PIP` – These are the output polygonal features that contain a point.

```

FACTORY_DEF Shape DonutFactory \
  INPUT FEATURE_TYPE ForestCoverPolygon \
  INPUT FEATURE_TYPE InteriorPoint \
  OUTPUT POLYGON FEATURE_TYPE ForestCoverPolygon \
  OUTPUT DONUT FEATURE_TYPE ForestCoverPolygon \
  OUTPUT POINT FEATURE_TYPE ForestCoverPolygonPoint \
  OUTPUT PIP FEATURE_TYPE ForestCoverPIP

```


DonutHoleFactory

Syntax

```
FACTORY_DEF <ReaderKeyword> DonutHoleFactory
[FACTORY_NAME <factory name>]
[INPUT FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*
[OUTPUT (OUTERSHELL|HOLE) FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*
```

Overview

This factory takes a feature that should be a donut polygon, a plain polygon, or an aggregate of such. It splits the aggregate or donut polygon into its components – outer shells and a series of holes – and outputs them accordingly. The holes are guaranteed to be output in the same order in which they are stored on the original feature. A single polygon is output untouched. Each component has all of the original feature’s attributes.

Assumptions

The DonutHoleFactory assumes that all features it gets are either single polygons, donut polygons, or aggregates of such.

Tip

The DonutHoleFactory is often used to split complex polygons into simple geometry for output to systems lacking support for polygons with holes.

Output Tags

The DonutHoleFactory supports the following output tags.

Tag	Description
OUTERSHELL	The outside containing ring(s) of the original aggregate or donut polygon are output with this tag.
HOLE	Each hole in the original feature is output with this tag.

Example

The following example uses a `DonutHoleFactory` to split complex polygons into their components. Notice how each original polygon is assigned a unique number that could be used to tie the holes back to their original polygon at a later time.

```

FACTORY_DEF Shape DonutHoleFactory \
INPUT FEATURE_TYPE ForestCoverPolygon \
    polygonNumber @Count(polygons) \
OUTPUT OUTERSHELL FEATURE_TYPE Shells \
OUTPUT HOLE        FEATURE_TYPE Holes

```

ElementFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> ElementFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [LIST_NAME <list name>{}]
  [ELEMENT_NUMBER_FIELD <field name>]
  [BASE_NUMBER_LIST_FIELD <field name>]
  [CLONE_GEOMETRY [yes|no]]
  [MODE (CLASSIC|LEAN|LEAN_AND_MEAN)]
  [OUTPUT (BASE|ELEMENT)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory takes a feature with a list identified by <list name> and returns one feature for each element in that list, as well as the original feature. The list elements are returned without geometry on them unless the CLONE_GEOMETRY clause is specified in the factory definition.

When CLONE_GEOMETRY is specified with no arguments, or with yes as its argument, each element feature has the geometry of the original feature. In this case, an attribute named `ElementFactory.baseCloned` is added to each feature output from the factory via the BASE output tag. If the output BASE feature has been cloned, then it will have the value of TRUE, otherwise the value will be FALSE.

If BASE_NUMBER_LIST_FIELD is specified, then this is taken to indicate the name of a list attribute that should be added to the BASE feature that is output. This list attribute will contain all the element numbers that were created from this feature, in order. This attribute may be useful in rejoining the elements later in a ReferenceFactory or through some other method.

The original feature is output if the BASE output tag is specified. If the ELEMENT_NUMBER_FIELD is specified, then each element feature output is given an attribute containing the element's list position.

The MODE directive allows you to fine-tune the type of attributes that will appear on the elements that are created. This is especially useful in improving efficiency when extremely large lists are being split apart. If no MODE is specified, the CLASSIC mode is used as the default.

Tip

The @XValue, @YValue, and @ZValue functions can be used to supply coordinates to element point features.

CLASSIC Mode

The list elements that are produced in `CLASSIC` mode will retain all the attributes of the original `BASE` feature. In addition, all of the attributes of each particular element in the list will be added without the list prefix.

LEAN Mode

The list elements that are produced in `LEAN` mode will retain all the attributes of the original `BASE` feature except for the list attributes referred to in the `LIST_NAME` clause. In addition, all of the attributes of each particular element in the list will be added without the list prefix.

LEAN_AND_MEAN Mode

The list elements that are produced in `LEAN_AND_MEAN` mode will not retain any of the attributes of the original `BASE`. They will only add the attributes of each particular element in the list, without the list prefix. When the lists have large numbers of elements, using this mode will significantly speed up performance.

Example:

Suppose we have a `BASE` feature with the following 6 attributes. (The `LIST_NAME` used here would be `"ID{ }"`.)

```
Name
Type
ID{0}.dec
ID{0}.hex
ID{1}.dec
ID{1}.hex
```

The directive `MODE CLASSIC` will produce elements with 8 attributes:

```
Name
Type
dec
hex
ID{0}.dec
ID{0}.hex
ID{1}.dec
ID{1}.hex
```

- *FME Functions and Factories*

dec
hex

dec
hex

None.

The `ElementFactory` supports the following output tags.

Tag	Description
BASE	The original feature.
ELEMENT	<p>A single feature for each element in the list.</p> <ul style="list-style-type: none"> • The attributes present on this feature are based on the <code>BASE</code> feature and depend on the <code>MODE</code> being used. • These features have no geometry unless the <code>CLONE_GEOMETRY</code> clause is specified in the factory definition. • The feature may have an additional field indicating its position in the original list.

The following example illustrates the use of the `ElementFactory` to break IGDS text multi-line objects apart into their components. This is needed when translating from IGDS to a system such as Shape, which cannot accommodate text multi-line objects.

```
FACTORY_DEF IGDS ElementFactory \
    INPUT FEATURE_TYPE 35 igds_type igds_multi_text \
    LIST_NAME igds_text_elements{} \
    OUTPUT ELEMENT FEATURE_TYPE 35 \
        igds_type igds_text_frag
```

The feature below matches the input specification of this factory definition and enters the factory:



Feature Type: 35

Attribute	Value
igds_type	igds_multi_text
igds_text_elements{0}.igds_text_string	Hello
igds_text_elements{0}.x	477556
igds_text_elements{0}.y	5360183
igds_text_elements{1}.igds_text_string	World
igds_text_elements{1}.x	477556
igds_text_elements{1}.y	5359177
Coordinates: (477553,5360181,20)	

Tip

Since no OUTPUT BASE clause is specified, the original feature input to the ElementFactory is deleted.

The factory splits the above feature into two output features that are shown as they emerge from the factory after the `OUTPUT ELEMENT` clause has been applied to them:



Feature Type: 35

Attribute	Value
igds_type	igds_text_frag
igds_text_string	Hello
x	477556
y	5360183
igds_text_elements{0}.igds_text_string	Hello
igds_text_elements{0}.x	477556
igds_text_elements{0}.y	5360183
igds_text_elements{1}.igds_text_string	World
igds_text_elements{1}.x	477556
igds_text_elements{1}.y	5359177

Tip

The shaded attributes have been added or changed. Notice that all original attributes are still present in each of the output elements because the default `CLASSIC` mode was used.



Tip

Applying the @XValue(&x) and @Yvalue(&y) functions to the element feature would provide it with coordinates that could be used by an output format.

ExtensionFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```
FACTORY_DEF <ReaderKeyword> ExtensionFactory
[FACTORY_NAME <factory name>]
[INPUT FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*
EXTENSION_LENGTH <length>
[SEGMENTS_TO_AVERAGE <segments>]
[OUTPUT (ORIGINAL|BEGINNING|END|STRETCHED)
  FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*
```

Overview

This factory accepts linear features and can output two-point linear features that extend the feature by a user-specified length. This factory can also output the original feature with the first and last segments stretched by a user-specified amount. Each of the created features gets a copy of all attributes of the original feature, including the feature type. Arcs that are input are converted to lines before processing.

One feature this factory can create is an extension of the first segment in the input feature. This feature's orientation is the same as the first segment of the input feature and its end point is the same as the input feature's start point. This new feature is output via the `OUTPUT BEGINNING` clause.

Another feature this factory can create is an extension of the last segment of the input feature. This feature's orientation is the same as the last segment of the input feature and its start point is the same as the input feature's end point. The feature holding this segment is output via the `OUTPUT END` clause.

The length of these extension features is controlled by the `EXTENSION_LENGTH` clause. This clause must specify a positive real number with a constant, or as an attribute value or as the output of a function. Each of the extension features consists of a two-point segment that has this length.

The optional `SEGMENTS_TO_AVERAGE` clause indicates the number of segments that should be considered when computing the orientation angle for the extension feature. By default, this is set to 1, which means the orientation of the extension feature matches the orientation of just one segment in the original feature. It can be set to any number of segments, in which case the orientation will be set to the average orientation of those segments. If the number of segments is larger than the number of segments available on the feature, then the entire feature orientation is averaged and used.

A third feature this factory can create is a duplicate of the input feature except that the first and last segments are extended in their respective orientation directions. The length of these extensions is also controlled by the `EXTENSION_LENGTH` clause. In this case the end nodes of the line are moved; no new nodes are added. The feature holding this segment is output via the `OUTPUT STRETCHED` clause.

All input features are output untouched via the `OUTPUT ORIGINAL` clause. In addition, any non-linear input features are immediately output without any extension features via the `OUTPUT ORIGINAL` clause.

If the `FME_GEOMETRY_HANDLING` directive is set to yes in the mapping file, arcs are extended as arcs instead of being stroked. The two point line segments are output normally, but the stretched version is converted to a path with three segments: the start line, the arc itself, and the end line.

Assumptions

None.

Output Tags

The `ExtensionFactory` supports the following output tags.

Tag	Description
BEGINNING	A two-point feature extending the input feature’s first segment.
END	A two-point feature extending the input feature’s last segment.
STRETCHED	A copy of the original feature, with the end points extended.
ORIGINAL	The original, unchanged input features.

Example

The following example takes parcel boundary features from a Shape file named `parcelbnds` and extends them before intersecting them against each other. This is in an attempt to correct for undershoots. The results are used to form polygons. Any unclosed lines that are shorter than the extension length are discarded, but lines remaining unclosed which are longer than this length are saved for inspection.

```
MACRO Tolerance 10
SHAPE_IN_DEF parcelbnds \
    SHAPE_GEOMETRY      shape_polyline \
    ID                  char(10)
SHAPE_OUT_DEF parcels \
    SHAPE_GEOMETRY      shape_polygon \
    FID                  char(10)
SHAPE_OUT_DEF dangles \
    SHAPE_GEOMETRY      shape_polyline \
    FID                  char(10)
FACTORY_DEF SHAPE_IN ExtensionFactory \
    FACTORY_NAME Extender \
    INPUT FEATURE_TYPE parcelbnds \
    EXTENSION_LENGTH $(Tolerance) \
    OUTPUT ORIGINAL FEATURE_TYPE * \
    OUTPUT BEGINNING FEATURE_TYPE * \
    OUTPUT END FEATURE_TYPE *
FACTORY_DEF SHAPE_IN IntersectionFactory \
    FACTORY_NAME Intersector \
    INPUT FEATURE_TYPE parcelbnds \
    OUTPUT SEGMENT FEATURE_TYPE *
FACTORY_DEF SHAPE_IN PolygonFactory \
    FACTORY_NAME Polygonizer \
    INPUT FEATURE_TYPE parcelbnds \
    END_NODDED \
    OUTPUT POLYGON FEATURE_TYPE parcels \
    OUTPUT LINE FEATURE_TYPE dangle
FACTORY_DEF SHAPE_IN TestFactory \
    FACTORY_NAME Weeder \
    INPUT FEATURE_TYPE dangle \
    TEST @Length() > $(Tolerance) \
    OUTPUT PASSED FEATURE_TYPE dangle
SHAPE_IN parcels
SHAPE_OUT parcels FID @Count(ParcelIds)
SHAPE_IN dangle
SHAPE_OUT dangles FID @Count(DangleIds)
```


GeoRSSFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> GeoRSSFactory \
[FACTORY_NAME <factory name>] \
[INPUT ENTRY FEATURE_TYPE <feature type> \
  [<attribute name> <attribute value>]* \
  [<feature function>]*]+ \
[INPUT FEED FEATURE_TYPE <feature type> \
  [<attribute name> <attribute value>]* \
  [<feature function>]*]? \
DATASET_ATTRIBUTE <attribute name> \
DIRECTION (READ|WRITE) \
[MERGE_ATTRIBUTES (TRUE|FALSE)] \
[BREAK_BEFORE_FIELD_CHANGE [<attribute name>]+] \
[BREAK_AFTER_FIELD_CHANGE [<attribute name>]+] \
[FAIL_IF_INVALID_GEORSS (TRUE|FALSE)] \
[FEATURE_NUMBER <attribute name>] \
[FEATURE_ID <attribute name>] \
[FEEDSTORE_ID <attribute name>] \
[READERMODE (NORMAL|UPDATE)] \
[MAX_ENTRY_AGE <value>]\
[FEEDSTORE_DB_DIR <value>]\
[PROXY_URL <value>] \
[PROXY_PORT <value>] \
[PROXY_PASSWORD <value> ]\
[PROXY_AUTH_METHOD <value>]\
[OUTPUT_FORMAT (Atom|RSS)] \
[GEOMETRY_OUTPUT_FORMAT (SIMPLE|W3C|GML)] \
[WRITER_CHARSET <value>] \
[ESCAPE_HTML (TRUE|FALSE)] \
OUTPUT (GEORSS|INVALID_GEORSS|FEED|ENTRY) FEATURE_TYPE \
<feature type> \
  [<attribute name> <attribute value>]* \
  [<feature function>]*]*

```

Overview

This factory is used to construct features from GeoRSS documents or feeds that are stored in an attribute of the input features or to construct GeoRSS documents from features. The **DIRECTION** clause specifies the factory's mode of operation. If it is used in **READ** mode, all input features are treated equally regardless of which input tag is used. If it is used in **WRITE** mode, the **FEED** input tag identifies the metadata that will be written out, and the **ENTRY** input tag identifies the entries in the document. The clauses **FEEDSTORE_ID** to **ESCAPE_HTML** are clauses for the GeoRSS Reader/Writer, and documentation on these clauses is to be found in the documentation for the GeoRSS Reader/Writer. Note however that the **FEEDSTORE_ID** attribute in particular has a slightly different meaning here than it does when used directly for the reader.

When the `DIRECTION` is `READ`, the factory constructs FME features from GeoRSS documents or feeds. The GeoRSS documents are expected to be stored in an attribute of the input features; the `DATASET_ATTRIBUTE` clause specifies the name of this attribute. The contents of this attribute should either be an xml-document conforming to the GeoRSS specification, the filename of a GeoRSS document, or a URL which points to a GeoRSS feed. Features from the GeoRSS document are output via the `OUTPUT FEED` and `OUTPUT ENTRY FEATURE_TYPE` clauses. The `INVALID_GEORSS FEATURE_TYPE` clause outputs features where the GeoRSS document/feed specified by the `DATASET_ATTRIBUTE` does not contain valid GeoRSS.

The factory also contains optional clauses that specify additional attributes for features output by the `OUTPUT GeoRSS FEATURE_TYPE` clause. The specification of the optional `MERGE_ATTRIBUTES` clause results in the copying of attributes from the input feature onto the GeoRSS features. Since a GeoRSS document may contain several features, the factory allows the sequence of GeoRSS features from a particular GeoRSS document to be numbered; the optional `FEATURE_NUMBER` clause can be used to specify the name of this attribute. If an attribute name is specified by the optional `FEATURE_ID` clause, then each feature constructed from the feed/document that was specified by an input feature will be tagged with the value that the input feature had for that attribute. For example, one might use a counter on incoming features, and set the `FEATURE_ID` clause to ‘_count’ to generate an identifier so that each output feature can be easily traced to the input feature which created it. The `FEEDSTORE_ID` clause is like the reader parameter of the same name, except that instead of taking a constant value, it takes the name of an attribute and for each input feature, the `FEEDSTORE_ID` is set to the value of the attribute on the input feature.

In the case that the GeoRSS document contains an error, the default behaviour of this factory is to halt the translation so that the user may correct the data if possible. If the clause `FAIL_IF_INVALID_GEORSS` is set to `FALSE`, then the feature which provided the invalid document will be output via the `INVALID_GEORSS` clause, and no features will have been extracted from the GeoRSS document. In either case, no features will be ever be constructed out of a document which contains errors.

When the `DIRECTION` is `WRITE`, the factory constructs GeoRSS documents from the input features and stores them in an attribute for the features that are output by the `OUTPUT GeoRSS FEATURE_TYPE` clause; the attribute to store the GeoRSS document is specified by the `DATASET_ATTRIBUTE` clause. The GeoRSS documents conform, where possible, to the type of document specified by the `OUTPUT_FORMAT` clause.

At any given moment, the factory, whether it is being used in `READ` or `WRITE` mode, is creating only one feature. When the factory is used in `READ` mode, all features constructed from the specified document/feed are output at the end of

the document. In case of error, no features from an erroring document/feed are produced. When the factory is used in WRITE mode, the current feature being constructed is output whenever any of the following is true:

- 1 The values of any of the `BREAK_BEFORE_FIELD_CHANGE` fields change from one feature to the next. When this occurs, the newly received input feature is not part of the flushed feature, but rather is part of the next feature to be constructed.
- 2 The values of any of the `BREAK_AFTER_FIELD_CHANGE` fields change from one feature to the next. When this occurs, the newly received input feature is part of the flushed feature.
- 3 There are no more input features left to be inserted into the constructed document.

Assumptions

The features entering the factory in the `READ` direction are assumed to contain an attribute that contains a valid GeoRSS document/feed as its value; this attribute must be identified by the `DATASET_ATTRIBUTE` clause. If the GeoRSS document is invalid, the translation will fail unless `FAIL_IF_INVALID_GEORSS` is set to `FALSE`.

The input features coming into the factory in the `WRITE` direction must have specific FME GeoRSS format attributes. It is incorrect to route more than a single feature through the `FEED` input tag. A warning will be logged, and only the first feature will be used to determine metadata for the output document(s).

The output tag `INVALID_GEORSS`, and the `FAIL_IF_INVALID_GEORSS` field are only used when the direction of the factory is set to `READ`. The `OUTPUT_FORMAT` and the `GEOMETRY_OUTPUT_FORMAT` clauses are obligatory when the direction is `WRITE`.

Output Tags

The `GeoRSSFactory` supports the following output tags.

Tag	Description
<code>GEORSS</code>	The features containing the GeoRSS document in the attribute specified by the <code>DATASET_ATTRIBUTE</code> in the <code>WRITE</code> direction.
<code>INVALID_GEORSS</code>	The features that triggered an error when extracting features from a GeoRSS document/feed. This is a subset of the features that entered the factory.
<code>FEED</code>	For any given GeoRSS document, the metadata for the feed.
<code>ENTRY</code>	The entry features from the input GeoRSS document/feed.

Example

The example below illustrates the usage of the GeoRSSFactory in the `WRITE` and `READ` directions. The first factory outputs features having an attribute called `TransactionData` that contains the GeoRSS documents created from the factory's input features. The second factory takes the output of the first factory and creates GeoRSS features out of the features from the first factory.

```

FACTORY * GeoRSSFactory                                     \
  INPUT FEATURE_TYPE * @Transform(SHAPE,GeoRSS)           \
  DATASET_ATTRIBUTE TransactionData                       \
  DIRECTION WRITE                                         \
  OUTPUT_FORMAT Atom                                     \
  GEOMETRY_OUTPUT_FORMAT SIMPLE                           \
  OUTPUT GEORSS FEATURE_TYPE newGeoRSSFeatures           \

FACTORY * GeoRSSFactory                                     \
  INPUT FEATURE_TYPE newGeoRSSFeatures                   \
  GeoRSS_DATA_ATTR TransactionData                       \
  DIRECTION READ                                         \
  FAIL_IF_INVALID_GEORSS FALSE                           \
  FEATURE_ID_id                                         \
  FEATURE_NUMBER_sequence_number                       \
  FEEDSTORE_DB_DIRfeedstores/GEORSS/today              \
  OUTPUTGEORSS FEATURE_TYPE *                           \
  OUTPUT INVALID_GEORSS FEATURE_TYPE *                  \
    @SupplyAttributes("bad_input","TRUE")

```

GML2Factory

Syntax

```

FACTORY_DEF <ReaderKeyword> GML2Factory                                \
  [FACTORY_NAME <factory name>]                                         \
  [INPUT FEATURE_TYPE <feature type>                                   \
    [<attribute name> <attribute value>]*                                \
    [<feature function>]*]*                                              \
  [GML2_DATA_ATTR <attribute name>]                                     \
  [DIRECTION (READ|WRITE)]                                              \
  [MERGE_ATTRIBUTES (TRUE|FALSE)]                                       \
  [MERGE_PREFIX<string prefix>]                                         \
  [BREAK_BEFORE_FIELD_CHANGE [<attribute name>]+]                       \
  [BREAK_AFTER_FIELD_CHANGE [<attribute name>]+]                       \
  [FAIL_IF_INVALID_GML (TRUE|FALSE)]                                    \
  [XFMAP <xfMap file>]+                                                 \
  [FEATURE_NUMBER <attribute name>]                                     \
  [FEATURE_ID <attribute name>]                                         \
  [OUTPUT (ORIGINAL|GML2|INVALID_GML) FEATURE_TYPE                   \
    <feature type>                                                       \
    [<attribute name> <attribute value>]*                                \
    [<feature function>]*]*

```

Overview

This factory is used to construct features out of GML2 documents that are stored in an attribute of the input features. It can also be used to construct GML2 documents from features. The **DIRECTION** clause specifies the factory's mode of operation.

When the **DIRECTION** is **READ**, the factory constructs FME features from GML2 documents. The GML2 documents are expected to be stored in an attribute of the input features; the **GML2_DATA_ATTR** clause specifies the name of this attribute. The original feature is output untouched via the **OUTPUT ORIGINAL FEATURE_TYPE** clause, while features from the GML2 document are output via the **OUTPUT GML2 FEATURE_TYPE** clause. The **INVALID_GML FEATURE_TYPE** clause outputs features whose **GML2_DATA_ATTR** clause contains invalid GML2.

The factory also contains optional clauses that specify additional attributes for features output by the **OUTPUT GML2 FEATURE_TYPE** clause. The specification of the optional **MERGE_ATTRIBUTES** and **MERGE_PREFIX** clauses results in the copying of attributes from the **ORIGINAL** feature into the **GML2** features. If **MERGE_PREFIX** is used, then the string prefix is put at the start of each attribute from the **ORIGINAL** feature. Since a GML2 document may contain several features, the factory allows the sequence of **GML2** features from a particular GML2 document to be numbered; the optional **FEATURE_NUMBER** clause can be used to specify the name of this attribute. Furthermore, each **GML2** feature can also be tagged by a unique sequence number that identifies them as coming from the same GML2 document; the optional **FEATURE_ID** clause can be used

to specify the name for this attribute. In case of an error in the GML2 document, if `FAIL_IF_INVALID_GML` is `FALSE`, no features will be extracted from the GML2 document, and the input feature will be output via the `INVALID_GML_FEATURE_TYPE` clause.

The optional `xfMAP` clauses specifies the `xfMaps` document to used. If this is not specified, then the `GML2Factory` assumes that the GML2 document contains features stored with the GML2 SAFE schema.

When the `DIRECTION` is `WRITE`, the factory constructs GML documents from the input features and stores them in an attribute for the features that are output by the `OUTPUT GML2 FEATURE_TYPE` clause; the attribute of the GML2 feature containing the GML2 document is specified by the `GML2_ATTR_DATA` clause. The GML2 documents written under the attribute that is specified by the `GML2_ATTR_DATA` clause conforms to the GML2 SAFE schema.

At any given moment, the factory, whether it is being used in `READ` or `WRITE` mode, is creating only one feature. The current feature being constructed is output whenever any of the following is true:

- 1 The values of any of the `BREAK_BEFORE_FIELD_CHANGE` fields change from one feature to the next. When this occurs, the newly received input feature is not part of the flushed feature, but rather is part of the next feature to be constructed. Values of attributes named in these clauses are transferred to the output features.
- 2 The values of any of the `BREAK_AFTER_FIELD_CHANGE` fields change from one feature to the next. When this occurs, the newly received input feature is part of the flushed feature. Values of attributes named in these clauses are transferred to the output features.

Assumptions

The features entering the factory in the `READ` direction are assumed to contain an attribute that contains a valid GML2 document as its value; this attribute must be identified by the `GML_DATA_ATTR` clause. If the GML2 document is invalid, the translation will fail unless `FAIL_IF_INVALID_GML` is set to `FALSE`.

The output tag `INVALID_GML`, and the `FAIL_IF_INVALID_GML` field are only used when the direction of the factory is set to `READ`.

Output Tags

The `GML2Factory` supports the following output tags.

Tag	Description
ORIGINAL	The original, unchanged input features.

Tag	Description
GML2	The features extracted from the GML2 document in the READ direction, or the features containing the GML2 document in the attribute specified by the GML_DATA_ATTR in the WRITE direction.
INVALID_GML	The features that triggered an error when extracting features from a GML2 document. This is a subset of the features output via the ORIGINAL output tag.

Example

The example below illustrates the usage of the GML2Factory in the WRITE and READ directions. The first factory outputs features having an attribute called TransactionData that contains the GML2 documents created from the factory's input features. The second factory takes the output of the first factory and creates GML2 features out of the features from the first factory.

```

FACTORY * GML2Factory                                     \
  INPUT FEATURE_TYPE *                                   \
  GML2_DATA_ATTR TransactionData                         \
  DIRECTION WRITE                                       \
  OUTPUT ORIGINAL FEATURE_TYPE *                       \
  OUTPUT GML2 FEATURE_TYPE myGMLFeatures

FACTORY * GML2Factory                                     \
  INPUT FEATURE_TYPE myGMLFeatures                     \
  GML2_DATA_ATTR TransactionData                         \
  DIRECTION READ                                       \
  FEATURE_IDgml_id                                     \
  FEATURE_NUMBERgml_number_in_id                       \
  OUTPUTGML2 FEATURE_TYPE *

```

GridToVectorFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```

FACTORY_DEF <ReaderKeyword> GridToVectorFactory           \
[FACTORY_NAME <factory name>]                             \
[AGGREGATE_OUTPUT (YES|NO)]                               \
[INPUT FEATURE_TYPE <feature type>                         \
  [<attribute name> <attribute value>]*                     \
  [<feature function>]* ]*                                  \
[OUTPUT (POINTS|ROWS|COLUMNS)                             \
  FEATURE_TYPE <feature type>                               \
  [<attribute name> <attribute value>]*                     \
  [<feature function>]*]*

```

Overview

This factory takes three-dimensional DEM raster features and extracts their elements as vector features. The factory is useful for converting from features of `fme_raster` geometry to features of `fme_point` or `fme_line` geometry.

The `AGGREGATE_OUTPUT` clause is used to specify that the output point or line features should be output as a single aggregate feature of points or lines. If this is the case, one aggregate feature will be output for each DEM raster feature input.

If the output tag `POINTS` is specified, then each cell or element in the raster is output as a 3D point feature.

If the output tag `ROWS` is specified, then each row of cells or elements in the raster is output as a 3D line feature.

If the output tag `COLUMNS` is specified, then each column of cells or elements in the raster is output as a 3D line feature.

See below for a description of the other types of output clauses that are supported.

Assumptions

None.

Output Tags

Tag	Description
POINTS	If specified, the input raster is output as a set of 3D point features through this clause. Each feature represents a z value at a row and column intersection point in the input raster.
ROWS	If specified, the input raster is output as a set of 3D line features through this clause. Each feature represents a row of z values in the input raster with vertices at column intersections.
COLUMNS	If specified, the input raster is output as a set of 3D line features through this clause. Each feature represents a column of z values in the input raster with vertices at row intersections.

IntersectionFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> IntersectionFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [OVERLAP_COUNT_ATTRIBUTE <attribute name>]
  [SEGMENT_COUNT_ATTRIBUTE <attribute name>]
  [LIST_NAME <list name>]
  [SELF_INTERSECTION_ONLY [(YES|NO)]]
  [SEPARATE_COLLINEAR_SEGMENTS [(YES|NO)]]
  [NODE_NUMBER_ATTR <attribute name>]
  [ATTRIBUTE_PREFIX_ATTR <attribute name>]
  [DISTINCT_ATTR <attribute name>]
  [IGNORE_NODE_HEIGHTS [(YES|NO)]]
  [GROUP_BY [<attribute name>+]*]
  [OUTPUT (SEGMENT|NODE|ILLEGAL_GEOM)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory takes a series of linear or polygonal features and inserts a node at each point where the input features cross. In addition, it removes any overlapping segments present in the input data. If desired, the attribute name given by the `OVERLAP_COUNT_ATTRIBUTE` clause will be added to the feature, and its value will be the number of input lines that overlapped on top of the output segment.

When nodes are added, if the original features had measures, the new points will have the same measures, interpolated between the nearest points by distance along the connecting segment. For adding points to arcs, the distance around the perimeter of the arc is used.

If `SELF_INTERSECTION_ONLY` is specified, then the factory only computes intersections on a feature-by-feature basis. Each feature is tested against itself and broken into pieces at each intersection point. No feature-to-feature comparisons are made. In this case, the value set to the `OVERLAP_COUNT_ATTRIBUTE` will be the number of features that result from removing self intersections. If the feature did not self-intersect, the attribute will be set to 1.

If the segment had several overlapping input features, the attributes of each of the input features will be added to the feature in the list identified by `<list name>`, if it was specified. In any case, each output feature is also assigned the

attributes of one of its original input features. The FME also writes in a “direction” attribute as it fills out the list in `LIST_NAME`, putting `same` if the geometry is oriented in the same direction, and `opposite` if the geometry is oriented in the opposite direction to the current geometry.

If `SEGMENT_COUNT_ATTR` is specified, an attribute named with the corresponding `<attribute name>` will be added to each output segment. This attribute contains the total number of output segments that correspond with the input feature from which this particular segment was obtained. If an input feature was broken into `n` output segments, each of those segments will have an attribute named `<attribute name>` which has a value of `n`.

This factory differs from the `ArcFactory` in that it inserts new nodes into the data in order to break features – the input does not need to be correctly noded.

If indicated, the node locations where intersections occurred are output as separate `NODE` features. If you would like each node to have a unique ID number, `NODE_NUMBER_ATTR` is the name of the attribute you want to store the ID number on the node feature itself.

If `ATTRIBUTE_PREFIX_ATTR` is specified, then a list attribute is added to each node feature that contains information on all of the line segments that intersect at that location. The information that is placed on this list includes the angle the line segments intersect the node, whether the segment is incoming to or outgoing from the node, and all the segment’s attributes. Here is an example of what would appear on a node feature if `attrInfo` is specified as the value for `ATTRIBUTE_PREFIX_ATTR`. (The order the segment information appears in the list is the order they intersect the node: counterclockwise from the right horizontal axis.)

attribute name	example value
<code>attrInfo{0}.fme_node_angle</code>	45
<code>attrInfo{0}.fme_node_direction</code>	outgoing
<code>attrInfo{0}.attributeA</code>	7
<code>attrInfo{0}.attributeB</code>	HWY 57
<code>attrInfo{0}.attributeC</code>	3.1
<code>attrInfo{1}.fme_node_angle</code>	90
<code>attrInfo{1}.fme_node_direction</code>	incoming
<code>attrInfo{1}.attributeA</code>	2
<code>attrInfo{1}.attributeB</code>	Pine Street
<code>attrInfo{1}.attributeC</code>	8.4

If `DISTINCT_ATTR` is specified, then not all line segments that intersect at that node will have the information listed in the format outlined above. The attribute listed here will be checked, and for each unique value of this attribute, one segment will be chosen to have its attributes listed. (This is most useful when you are not interested in having the information listed for each intersected line segment, but only the information from the unbroken “original” lines. In this case, specify an attribute that has a unique value for all input lines, and only one representative segment for each “original” line will have its information listed.)

The creation of nodes can be calculated in 3D, if requested. Constructing nodes in 3D would mean that line segments would only share a node if they shared the same Z value at the point they intersected. Constructing nodes in 2D would mean that all intersecting segments would share a common node, regardless of their respective Z values. Consider, for example, a situation where two lines (which crossed) represented roads, where one road was an overpass above the other road. Suppose these two lines had differing elevations. If you constructed nodes in 3D, these two roads would not be linked to the same node where they crossed. Two nodes would be produced at the crossing point – each one with a different Z value. If you constructed nodes in 2D, these lines would both link to a common node, which would be present at the location where they crossed. In either the 2D or 3D case, the full dimensionality of the input is preserved in the output – 3D features are never converted to 2D. The 2D or 3D choice only indicates how the nodes are created and which lines are linked to them; it does not affect the dimension of the features that are output.

The `IGNORE_NODE_HEIGHTS` directive specifies how the topology will be computed. If it is set to `YES`, the topology will be computed in 2D. If it is `NO`, the topology will be computed in 3D. The default is `YES`.

This factory relies on a spatial index grid to optimize the search for intersections. The settings of this underlying grid can typically be ignored, as reasonable default values are used in all cases. However, the user may change any of the settings, detailed in the following paragraphs, based on input data specific properties, if desired.

If the `SEPARATE_COLLINEAR_SEGMENTS` clause is specified, any line segments which are collinear are not merged together; instead, one copy is output for each original feature which shared that segment. Each such segment will have the respective original feature’s attributes as its main attributes, and attributes from all other collinear features will be added as a list attribute, if the `LIST_NAME` clause is given.

If the `FME_GEOMETRY_HANDLING` directive is set to “yes” in the mapping file, arcs that exist in the input (as arcs or ellipses) will be preserved as arcs in the output; these will otherwise be stroked to lines before intersections are processed.

Assumptions

None.

Output Tags

The `IntersectionFactory` supports the following output tags.

Tag	Description
ILLEGAL_GEOM	Any original input features that were points, NULL geometry, etc. are immediately output via this tag. Normally, this is not specified.
SEGMENT	The intersected features are output by this tag. If the <code>LIST_NAME</code> was specified, these features will have an attribute containing their number of overlapping input features. They will also have all attributes of the original features.
NODE	The locations of every intersection are represented by point features and are output by this tag. The <code>NODE_NUMBER_ATTR</code> , <code>ATTRIBUTE_PREFIX_ATTR</code> , and <code>DISTINCT_ATTR</code> keywords modify which attributes are included on these output features.

Example 1

The following example illustrates the use of the `IntersectionFactory` to compute all intersections in the input data and remove all duplicate segments. In addition, each output segment also has an attribute indicating the number of input features that overlapped it.

```

FACTORY_DEF SHAPE IntersectionFactory      \
  INPUT FEATURE_TYPE roads                  \
  OVERLAP_COUNT_ATTRIBUTE numOverlaps      \
  LIST_NAME originalAttrs{}                \
  OUTPUT SEGMENT FEATURE_TYPE roadSegments

```

Example 2

In this example, the `IntersectionFactory` is first used to split any self-intersecting lines into non-self-intersecting pieces, and then the results are intersected against each other. This is necessary to ensure entirely *clean* output.

```

FACTORY_DEF SHAPE IntersectionFactory      \
  INPUT FEATURE_TYPE roads                  \
  SELF_INTERSECTION_ONLY                   \
  OUTPUT SEGMENT FEATURE_TYPE roadSegments
FACTORY_DEF SHAPE IntersectionFactory      \
  INPUT FEATURE_TYPE roadSegments          \
  OUTPUT SEGMENT FEATURE_TYPE roadSegments

```


JSONQueryFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> JSONQueryFactory \
[FACTORY_NAME <factory name>] \
[INPUT FEATURE_TYPE <feature type> \
  [<attribute name> <attribute value>]* \
  [<feature function>]*]* \
MODE (EXTRACT|EXPLODE) \
EXPLODE_FORMAT (JSON|GEOJSON|ESRIJSON) \
EXTRACT_ATTR (Yes|No) \
JSON_ATTR <attribute name> \
EXPLODE_QUERY <JSON query> \
QUERIES [<attribute name> <JSON query>]+ \
OUTPUT (EVALUATED|EXPLODED) \
  FEATURE_TYPE <feature type> \
  [<attribute name> <attribute value>]* \
  [<feature function>]*]*

```

Overview

This factory uses JSON queries to extract values from JSON text stored in a feature attribute, or to convert JSON text into new features. The **JSON_ATTR** clause specifies the feature attribute that contains the JSON text, and the **MODE** clause specifies the factory's mode of operation. The **EXPLODE_FORMAT** clause specifies if the JSON text being converted is in plain JSON, GeoJSON, or ESRIJSON format. If it is in GeoJSON or ESRIJSON, then the exploded features will be treated as GeoJSON or ESRIJSON features that have attribute and geometry information.

JSON Queries

A JSON query is a simple mechanism to refer to values within a JSON document. A query is made up of one or more expressions, which are separated by a + operator. There are three types of expressions: JSON structure expressions, JSON property expressions and string literal expressions.

JSON Structure Expressions

A JSON structure expression can refer to values in a JSON document. The outermost JSON element, which must be an array or an object, is always referred to by the term *json*, and this term must appear at the beginning of every JSON structure expression. The child elements can be referred to using JavaScript-like square bracket index operators. For example, if the outermost element is an array, the first element of the array can be referred to by the expression *json[0]*, the second element can be referred to by the expression *json[1]*, and so on. Likewise, if the outermost JSON element is an object, with

keys “name” and “address”, then the values of these keys can be referred to by the expressions `json[“name”]` and `json[“address”]` respectively.

These index operators can be cascaded. For example, if the outermost JSON element is an object with a key and “address” whose value is an object containing keys “city” and “province”, then these values can be referred to by the expressions `json[“address”][“city”]` and `json[“address”][“province”]`.

In order to refer to all of the values in an array or object, a wildcard index `*` can be used. For example, if the outermost JSON element is an array, then the expression `json[*]` refers to every element in the array.

JSON Property Expressions

A property expression is a structure expression as described above, followed by a `.` (dot) operator and a property name. Currently, the only supported properties are *type* and *size*. The type property returns the type of the JSON value referred to by the JSON structure expression. For example, if the outermost JSON element is an array, and the first element of the array is a string, then the expression `json[0].type` would have a value of *string*. The size property, which can only be applied to an array, returns the number of elements in the array.

String Literal Expressions

A string literal expression is simply a quoted string value, such as *“this is a string literal expression”*.

Extract Mode

If the `MODE` clause is set to `EXTRACT`, the factory will expect the value of the `QUERIES` clause to be a list of attributes and JSON queries (ie. `attr1 query1 attr2 query2`). For each input feature, the factory will then apply each query to the JSON text found in the value of the attribute identified by the `JSON_ATTR` clause, and populate the query’s corresponding feature attribute with the value of the query.

The factory will log an error and then ignore any query evaluation errors, rather than abort the translation.

Explode Mode

If the `MODE` clause is set to `EXPLODE`, the factory will expect the value of the `EXPLODE_QUERY` clause to be a valid JSON query, which only contains structure type expressions. For each input feature, the factory will then apply the query to the JSON text found in the value of the attribute identified by the `JSON_ATTR` clause. For each JSON value referred to by the query, the factory will create a new FME feature which will contain the JSON value in the attribute identified by the `JSON_ATTR` clause. `EXPLODE_FORMAT` specifies how the JSON text should be parsed. If it is set to `‘JSON’`, the text will

be treated as plain JSON text. If it is set to 'GEOJSON' or 'ESRIJSON', GeoJSON or ESRIJSON features will be constructed from the JSON text. If no valid GeoJSON or ESRIJSON features are constructed, a warning will be issued and the text will be treated as plain JSON. The `EXTRACT_ATTR` clause specifies whether or not the keys of a JSON Object should be added to the new FME feature as attributes. If `EXTRACT_ATTR` is set to 'Yes' and `EXPLODE_FORMAT` is 'GEOJSON' or 'ESRIJSON', no attributes will be constructed from keys that are valid GeoJSON or ESRIJSON keys, as these will be handled by the respective format parser.

Each new FME feature produced from the JSON text contained by an input feature will contain a copy of all the attributes from the original input feature, as well as several new attributes.

The attribute identified by the `JSON_ATTR` clause will contain the JSON text that the feature represents. The `json_type` attribute of the new feature will contain the type of JSON value that the new feature was constructed from, and the `json_index` attribute will contain the index (array location or object key) of the JSON value that the feature represents.

The factory will log a warning, and then ignore any evaluation errors in the JSON query, rather than abort the translation.

Output Tags

The `JSONQueryFactory` supports the following output tags.

Tag	Description
EVALUATED	When the factory is run in <code>EXTRACT</code> mode, the FME features containing the new attributes will be output through the <code>EVALUATED</code> output tag.
EXPLODED	When the factory is run in <code>EXPLODE</code> mode, the FME features containing the new attributes will be output through the <code>EXPLODED</code> output tag.

Examples

The examples below illustrate the usage of the `JSONQueryFactory` in `EXTRACT` and `EXPLODE` modes.

The first example uses the factory in `EXTRACT` mode, and populates three feature attributes with the values of JSON queries. The third query demonstrates the use of a query with multiple expressions, including the use of a string literal expression.

```

FACTORY * JSONQueryFactory          \
      INPUT FEATURE_TYPE JSONFeature \
      MODE EXTRACT                   \
      JSON_ATTR json_text            \

```

```

    QUERIES lat "json[\"latitude\"]" long "json[\"longitude\
    \"]" name_and_description "json[\"name\"] + \" - \" + json[\"
    description\"]          \
    OUTPUT EVALUATED FEATURE_TYPE EvaluatedFeatures

```

The second example uses the factory in EXPLODE mode. It examines the JSON text in the `json_text` attribute and applies the query `json[“features”][*]` to the text. A new feature will be created for each JSON value referred to by the query.

```

FACTORY * JSONQueryFactory
INPUT FEATURE_TYPE JSONFeature
MODE EXPLODE
EXPLODE_FORMAT JSON \
EXTRACT_ATTR No \
JSON_ATTR json_text
EXPLODE_QUERY "json[\"features\"][*]"
OUTPUT EXPLODED FEATURE_TYPE ExplodedJSONFeature

```

LabelFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```

FACTORY_DEF <ReaderKeyword> LabelFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [PLACEMENT (UPPER_RIGHT|UPPER_LEFT|LOWER_RIGHT| LOWER_LEFT|
    RIGHT|LEFT)]
  [PLACEMENT_DISTANCE <distance from line>]
  [PLACEMENT_SPACING <distance along line>]
  [MINIMUM_LENGTH <length>]
  [LABEL_ENDS [<boolean value>]]
  [LABEL_FIELD <attribute name>]
  [LABEL_HEIGHT <label height in real world coords>]
  [AVERAGE_CHAR_WIDTH <avg. char. width in real world coords>]
  [FLUSH_LABELS <boolean value>]
  [OUTPUT (POINT|ORIGINAL)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory accepts linear and polygonal features and outputs one or more label point features that are spaced along the line as specified by `PLACEMENT_SPACING` clause. If `PLACEMENT_SPACING` is not specified then a single point is generated at the midpoint of the linear feature as long as `LABEL_ENDS` is set to no. The value may be specified as a constant, or as an attribute value or as the output of a function.

The output point features have all the attributes of the associated input linear feature from which they were derived as well as a `LabelRotation` attribute that contains the rotation of the label point. This rotation is adjusted so that text oriented at the label point will be parallel to the line segment and will not be upside down or right-to-left. In addition, the output point features have a `ParallelRotation` attribute which contains the true orientation of the line at the label point. This rotation can be used to place a symbol that indicates the direction of the line.

The output label point features are a `PLACEMENT_DISTANCE` from the closest point on the line. If `PLACEMENT_DISTANCE` is not specified then the point(s) are located on the linear feature. The value may be specified as a constant, or as an attribute value or as the output of a function.

When generated label points are a `PLACEMENT_DISTANCE` measured perpendicularly from the closest point on the line. The placement points are placed relative to the line as specified by the `PLACEMENT` clause. If `UPPER_RIGHT` is specified, then the points are located a `PLACEMENT_DISTANCE` above (or to the right if the line is vertical) the line. `UPPER_LEFT` is nearly identical except that if the line is vertical, the label point will be to the left of the linear feature. `LOWER_LEFT` and `LOWER_RIGHT` work similarly with all label points being below the linear feature unless it is vertical in which case the label points are to the left and right for `LOWER_LEFT` and `LOWER_RIGHT`, respectively. If `RIGHT` is specified then the text is to the `RIGHT` of the line. If `LEFT` is specified then the text is to the `LEFT` of the line.

If the optional `MINIMUM_LENGTH` clause is specified, it indicates the minimum length line which will have a label generated for it. If the line's length is less than this minimum, then no label will be generated for it. The value may be specified as a constant, or as an attribute value or as the output of a function.

If the optional `LABEL_ENDS` clause is specified, label points will be generated at the start and end point of the feature. If it is not specified, no label points are generated at the feature ends. The value may be specified as a Boolean constant, or as an attribute value or as the output of a function.

The optional `LABEL_FIELD`, `LABEL_HEIGHT`, and `AVERAGE_CHAR_WIDTH` clauses are used to output label points whose labels do not overlap.

The LabelFactory uses the `LABEL_HEIGHT` and `AVERAGE_CHAR_WIDTH` to calculate the `LABEL_FIELD`'s bounding box. The LabelFactory uses these label's bounding boxes to output non-overlapping labels. These values may be specified as constants, as attribute values or as the output of functions. If the `AVERAGE_CHAR_WIDTH` is not specified or its final value is 0.0, then the `LABEL_HEIGHT` value will be taken as the value for the `AVERAGE_CHAR_WIDTH`.

The optional `FLUSH_LABELS` clause allows input features to carry a Boolean attribute that will indicate if the factory needs to flush its current indexed labels. These indexed labels are kept by the factory to calculate label overlap. If specified, the value for this attribute should NOT be a Boolean constant; instead it should be specified as an attribute value or as the output of a function.

Any non-linear features which are input into the factory are output via the `ORIGINAL` tag untouched. All input features are also output untouched via the `ORIGINAL` tag.

If the `FME_GEOMETRY_HANDLING` directive is set to yes in the mapping file, arcs and ellipses are accepted as linear and polygonal features, thus producing label point features; otherwise, they are considered non-linear, and no label point features will be produced.

Assumptions

None.

Output Tags

The `LabelFactory` supports the following output tags.

Tag	Description
ORIGINAL	The original features are all output here regardless of input geometry type. Only linear and polygonal features are used to generate label points.
POINT	The label point features are output here. Each label point feature has all attributes and feature type of the linear feature from which it was generated with the new attribute <code>LabelRotation</code> being added so that when used to place text, the text is parallel to the linear feature and adjusted to read in a pleasant manner. In addition, the feature will have a <code>ParallelRotation</code> attribute which contains the true orientation of the feature at the label point. This rotation can be used to place a symbol rather than text.

Example

In the following example, label points are created for each linear or polygonal feature passing through the FME. Any feature that is not linear or polygonal is simply output via the `ORIGINAL` output tag. The label point features are created every 1500 units along the feature and are offset from the associated line 500 units perpendicular to the closest point on the line. Each point feature output has all of the attributes of the original feature as well as `LabelRotation` and `ParallelRotation` attributes.

```

FACTORY_DEF * LabelFactory                                \
  INPUT FEATURE_TYPE *                                    \
  PLACEMENT UPPER_LEFT                                     \
  PLACEMENT_DISTANCE 500                                   \
  PLACEMENT_SPACING 1500                                  \
  OUTPUT POINT FEATURE_TYPE *                             \
  OUTPUT ORIGINAL FEATURE_TYPE *
```


ListFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> ListFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [GROUP_BY [<attribute name>+]*]
  [LIST_NAME <list name>]
  [OUTPUT (LIST|SINGLETON)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory takes a series of features matching an input specification and joins them together based on attribute values specified by the `GROUP_BY` clause. One feature is output for each group resulting from the `GROUP_BY` clause. If no `GROUP_BY` clause is specified, then all features fall into the same group and a single feature is output.

The features output from the `ListFactory` have all of their attributes stored within the list identified by `<list name>`. There is one element in the list for each feature that is in the group from which the list is constructed. If no `GROUP_BY` clause is specified, then the number of elements in the list is equivalent to the number of features input into the `ListFactory`.

The output features have **no** geometry. If the `OUTPUT SINGLETON` clause is specified, features that are the only members of their group are output unchanged.

Assumptions

None.

Output Tags

The `ListFactory` supports the following output tags.

Tag	Description
<code>LIST</code>	The features for which lists are constructed. If the <code>SINGLETON</code> tag is specified, then each of these features has a list with at least two elements.
<code>SINGLETON</code>	Applies to features that are the only members in a group. If this isn't specified, then such features are output using the <code>LIST</code> tag with all their attributes stored in an element list having only one element. If an <code>OUTPUT SINGLETON</code> clause is specified, such features are output unchanged.

Example

The following example illustrates the use of the `ListFactory` to merge text features read from a Shape file into a single multi-text-line object. This is needed when translating to systems, like IGDS or SAIF, that handle complex multi-line text. For example, systems such as Shape can only group text elements together by common attribute values.

The definition of the `ListFactory` is shown below. Notice that as features enter the factory, the `@XValue` and `@YValue` functions are used to extract their coordinates and assign them to attributes, which are output in the list.

```

FACTORY_DEF Shape ListFactory                                \
  INPUT FEATURE_TYPE text    x @XValue()                    \
                                y @YValue()                  \
  GROUP_BY TEXTGROUP                                               \
  LIST_NAME text_elements{}                                         \
  OUTPUT LIST FEATURE_TYPE text_multi
```

Tip

Since no `OUTPUT SINGLETON` clause is specified, groups that have only one element will still be output as a list.

Feature Type: text

Attribute Name	Value
TEXTSTRING	Hello
TEXTGROUP	12
Coordinates: (477553,5360181,20)	

LLOutlineAndCentroidFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> LLOutlineAndCentroidFactory
[FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]+
  DATA_STRING <attribute name>
  DATA_FILE <attribute name>
  INDEX_FILE <attribute name>
  [BITFLAG_STRING <attribute name>]
  [OUTPUT (POINT|POLYGON|INVALID)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory uses a provided DLS, NTS or FPS data string and an optional bitFlag string to calculate the outline and the latitude-longitude centroid of a designated area. If the BITFLAG_STRING parameter is present, and if the value of the attribute designated by <attribute name> is a non-empty binary string whose length matches the requirements of the data string type, the function will use this bitFlag string to calculate the area differently.

Moreover, the factory uses the attributes specified by DATA_FILE and INDEX_FILE to initialize the dataset to be processed. For this reason, all of the DATA_STRING, DATA_FILE and INDEX_FILE parameters are required. Furthermore, any features missing the attributes specified by these parameters will be considered invalid and output without processing by the factory.

For all valid features, the factory starts by initializing the dataset specified by the corresponding attributes. If this step is successful, the factory then proceeds to determining the outline of the area specified by either the data string alone, or the data string and a valid bitFlag string. From this area's representation in the UTM projection, the factory determines the centroid.

Finally, if all these steps have been successful, the factory outputs two features. The first feature is a point representing the centroid of the area in latitude-longitude locations. This point feature has attributes with information about the vertices of the polygon representing the area's outline. The second feature is a polygon feature representing the outline of the calculated area. This polygon also has attributes indicating the latitude-longitude location of the centroid that was calculated from the area.

Output Tags

The `LLOutlineAndCentroidFactory` supports the following output tags.

Tag	Description
POINT	The factory uses clauses with this tag to output a point feature representing the centroid calculated from the specified area. This feature will include the <code>DATA_STRING</code> , <code>DATA_FILE</code> and <code>INDEX_FILE</code> attributes from the input feature, as well as a series of new numbered attributes named <i>PolygonVertex_X</i> and <i>PolygonVertex_Y</i> representing the coordinates of the vertices on the outline of the corresponding area.
POLYGON	The factory uses clauses with this tag to output a polygon feature representing the calculated area. This feature will include the <code>DATA_STRING</code> , <code>DATA_FILE</code> and <code>INDEX_FILE</code> attributes from the input feature, as well as a pair of new attributes named <i>centroidX</i> and <i>centroidY</i> representing the latitude-longitude coordinates of the centroid of the area.
INVALID	Any input features missing one of the attributes specified by the <code>DATA_STRING</code> , <code>DATA_FILE</code> or <code>INDEX_FILE</code> parameters will not be processed by the factory and output directly using clauses with this tag. Moreover, if all attributes are present but the factory encounters an error while processing the input feature, the original feature will be output using clauses with this tag.

Example

The first example below presents a situation where Shape features are being used for factory input. Note that the optional `BITFLAG_STRING` parameter is not specified, so the features output will be calculated exclusively from the provided data strings.

```

FACTORY_DEF SHAPE LLOutlineAndCentroidFactory
INPUT FEATURE_TYPE zones
DATA_STRING dataStringAttr
DATA_FILE dataFileAttr
INDEX_FILE indexFileAttr
OUTPUT POINT FEATURE_TYPE *
OUTPUT POLYGON FEATURE_TYPE *
OUTPUT INVALID FEATURE_TYPE *

```

MatchingFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> MatchingFactory
[FACTORY_NAME <factory name>]
[INPUT_FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*
[MATCH_GEOMETRY (2D|3D|FULL|NONE)]
[LENIENT_GEOMETRY_MATCH [(yes|no)]]
[EXTRA_VERTEX_TOLERANCE <tolerance>]
[INTERIOR_VERTEX_TOLERANCE <tolerance>]
[MATCH_ATTRIBUTES [(<attribute name>|fme_regexp_match
  <regexp>)]*]
[MATCH_ALL_ATTRS_EXCEPT [(<attribute name>|
  fme_regexp_match <regexp>)]*]
[MATCH_ALL_ATTRIBUTES [(yes|no)]]
[DIFFERING_ATTRIBUTES [(<attribute name>|fme_regexp_match
  <regexp>)]*]
[BLANK_AND_MISSING_DIFFER [(yes|no)]]
[ADD_TO_MATCHED [<attribute name> <attribute value>]+]
[SINGLE_MATCHED_LIST_NAME <list name>]
[OUTPUT (MATCHED|NOT_MATCHED|SINGLE_MATCHED|
  FAILURE_POINT)
  FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*

```

Overview

This factory can be used to detect and flag features that are matches of each other. Features are declared to match when they have matching geometry and matching attribute values. The specification of the factory determines how matches are declared.

This factory is often used with the Multi-Reader to identify features that two files have in common. In this way, the factory can also be used to perform change detection between the input files. In conjunction with the Multi-Reader, it is able to identify all features that two input files have in common, and those which are in one file and not the other, such as the additions and the deletions.

Assumptions

The geometry matching considers the geometries of two polygons, donuts, or aggregates to be equivalent even if they differ in the starting point of their rings or the order of their components.

- *FME Functions and Factories*

Clauses

The `MatchingFactory` makes use of several optional clauses to specify its operation.

Clauses	Description	Optional
MATCH_GEOMETRY (2D 3D FULL NONE)	<p>Controls if 2D, or 3D, or no geometry must be the same before a match is declared. Full may be set so that 3D, measures and geometry attributes must all match.</p> <p>When comparing raster geometries: 2D matches the properties, 3D matches the properties and values and FULL matches the properties, values and geometry attribute.</p> <p>When comparing surface or solid geometries: 2D behaves the same way as 3D, that is, Z values are always compared.</p> <p>Default: None</p> <p>Example: MATCH_GEOMETRY 2D</p>	Yes
LENIENT_GEOMETRY_MATCH [(yes no)]	<p>Controls how the linear and area geometries will be compared. If this flag is set to yes, the orientation direction and start point of the rings will be ignored when comparing area geometries. For linear features, the direction of the feature will be ignored while making comparisons. When this clause is set to no, all geometries are compared strictly (that is, they are compared coordinate by coordinate).</p> <p>When comparing raster geometries only the extents are compared. This factory returns an error if input contains a surface or solid geometry and this flag is set to yes as it is not supported yet.</p> <p>Default: no</p>	Yes

Clauses	Description	Optional
INTERIOR_VERTEX_TOLERANCE <tolerance>	<p>When geometry is being matched, this controls how close together interior vertices must be in order for them to be declared a match. Note that the start and end points of features <i>must</i> be identical for a match to be declared. The value is measured in ground units of the feature. This value is ignored when matching surface and solid geometries.</p> <p>Default: 0</p> <p>Example: INTERIOR_VERTEX_TOLERANCE 2.0</p>	Yes
EXTRA_VERTEX_TOLERANCE <tolerance>	<p>When geometry is being matched, this controls how close any extra vertices in a feature must be to the line connecting the adjoining matching vertices. This value is ignored when matching surface and solid geometries.</p> <p>Default: None</p> <p>Example: EXTRA_VERTEX_TOLERANCE 1</p>	Yes
MATCH_ATTRIBUTES [(<attribute <i>fme_regexp_match</i> <regexp>)]*	<p>Controls which attributes that are part of the input feature must have the same value before a match is declared. Attribute names may be listed explicitly or through a regular expression preceded with the special <i>fme_regexp_match</i> token. The values from all attributes matching the regular expression are concatenated together and compared to determine a match. See <i>RE Construction</i> on page 216 in the <i>@RemoveAttributes</i> section.</p> <p>Default: No Attributes</p> <p>Example: MATCH_ATTRIBUTES type</p>	Yes

Clauses	Description	Optional
ADD_TO_MATCHED [<attribute name> <attribute value>]+	Contains a series of attribute names and values to be added to each matched feature. Note: If the value was a function, it will only be invoked once and the same return value will be added to each matched feature. This can be used to assign a unique ID to each group of matching features, as the following example shows. Default: None Example: <pre>ADD_TO_MATCHED match_id @Count (Matches)</pre> If features a, b, and c all match, then a, b, and c will have the same value for match_id, and no other features will have this value.	Yes
SINGLE_MATCHED_LIST_NAME <list name>	If specified, a list with the given name is created on each feature output by the SINGLE_MATCHED tag. The contents of the list are the attributes of all input features corresponding to a given output feature.	Yes

Output Tags

The MatchingFactory supports the following output tags.

Tag	Description
MATCHED	Features found to have matches are output using this tag.
SINGLE_MATCHED	The first feature from each matching group of features are output using this tag. A copy of the same feature is also output via the MATCHED tag. This is useful when only one feature from a matching group is desired. The attributes from all features are also copied to features output via this clause.
NOT_MATCHED	Features found to have no matches are output using this tag.
FAILURE_POINT	Point features marking where features with matching attributes and start points diverge are output via this tag. The geometry of these features is the point at which the divergence began.

Example

In this example, the River features from two SAIF files, an original file and an updated version of the original, are compared using the MatchingFactory to determine the changes made.

The example shows how the factory can be used to determine features that are the same, features that have been deleted, and features that have been added during the update process.

```
#-----
# Set up for a multi-read of all the rivers and streams in two
# SAIF datasets
READER_TYPE MULTI_READER
MULTI_READER_TYPE{0} SAIF
MULTI_READER_KEYWORD{0} SAIF{0}
SAIF{0}_DATASET c:\saifdata\original.zip
SAIF{0}_IDS RiverStreams
MULTI_READER_TYPE{1} SAIF
MULTI_READER_KEYWORD{1} SAIF{1}
SAIF{1}_DATASET c:\saifdata\updated.zip
SAIF{1}_IDS RiverStreams
#-----
# This factory looks for matches among all the river objects
# read in that have "Arc" geometry (which in SAIF is the same as
# saying linear geometry). Only the x and y coordinates are
# compared when looking for matches, and the "type" attribute
# must have the same value for features to be matched. As well,
# the multi_reader_id attribute must be different -- this
# ensures that if the original file had two equivalent features,
# they are not matched.
# A "match_id" field is also added. It will contain the same
# value in any features which are declared as matched, though in
# this example, this value is never used.
FACTORY_DEF MULTI_READER MatchingFactory \
INPUT FEATURE_TYPE River::TRIM position.geometry.Class Arc \
MATCH_GEOMETRY 2D \
MATCH_ATTRIBUTES type \
DIFFERING_ATTRIBUTES multi_reader_id \
ADD_TO_MATCHED match_id @Count(matches) \
OUTPUT MATCHED FEATURE_TYPE * matched YES \
OUTPUT NOT_MATCHED FEATURE_TYPE * matched NO \
#-----
# Now set up to output the features to shape files
WRITER_TYPE SHAPE
#-----
# This Shape file gets all the rivers in the original SAIF file
# that have not been changed. This means that they were matched
# and so lived in both the first and second file. So we need only
# to save the ones from the first file, those from the second file
# can be ignored.
SHAPE_DEF unchangedRivers RIVERTYPE char(30)
SAIF River::TRIM matched YES multi_reader_id 0 type %type
SHAPE unchangedRivers RIVERTYPE %type
#-----
# This Shape file gets all the rivers in the second SAIF file
# that were not in the first. This means they were not matched and
# have a multi_reader_id of 1. Such features are ones which have
# been "added"
SHAPE_DEF newRivers RIVERTYPE char(30)
SAIF River::TRIM matched NO multi_reader_id 1 type %type
SHAPE newRivers RIVERTYPE %type
#-----
```

```
# This Shape file gets all the rivers in the first SAIF file
# that were not in the second. This means they were not matched and
# have a multi_reader_id of 0. Such features are ones which have
# been "deleted".
SHAPE_DEF deletedRivers RIVERTYPE char(30)
SAIF River::TRIM matched NO multi_reader_id 0 type %type
SHAPE deletedRivers RIVERTYPE %type
```

MRFCleanFactory

Note: MRFCleanFactory requires an extra-cost plug-in.

Syntax

```

FACTORY_DEF <ReaderKeyword> MRFCleanFactory2D |
MRFCleanFactory3D \
  [FACTORY_NAME <factory name>] \
  [AGGREGATE_OUTPUT (YES|NO)] \
  [INPUT FEATURE_TYPE <feature type>\
    [<attribute name> <attribute value>]* \
    [<feature function>]* ]* \
  [GROUP_BY <field_name>] \
  TOLERANCE <value> \
  [TOLERANCE_ATTR <attribute_name>] \
  [SIMPLIFY [(YES|NO)]] \
  [SHORT_ELEMENT [(YES|NO|FLAG)]] \
  [EXTEND [(YES|NO)]] \
  [INTERSECT [(YES|NO|FLAG)]] \
  [FUZZY_INTERSECT [(YES|NO)]] \
  [DUPLICATE_REMOVE [(YES|NO)]] \
  [JOIN [(YES|NO)]] \
  [CONFLATE [(YES|NO)]] \
  [DANGLER [(YES|NO|FLAG)]] \
  [OBJECT_CLEAN [(YES|NO)]] \
  DANGLEFACTOR [<value>] \
  FILTERFACTOR [<value>] \
  [OUTPUT (CLEANED|FLAGGED) FEATURE_TYPE <feature type> \
    [<attribute name> <attribute value>]* \
    [<feature function>]*]*

```

Overview

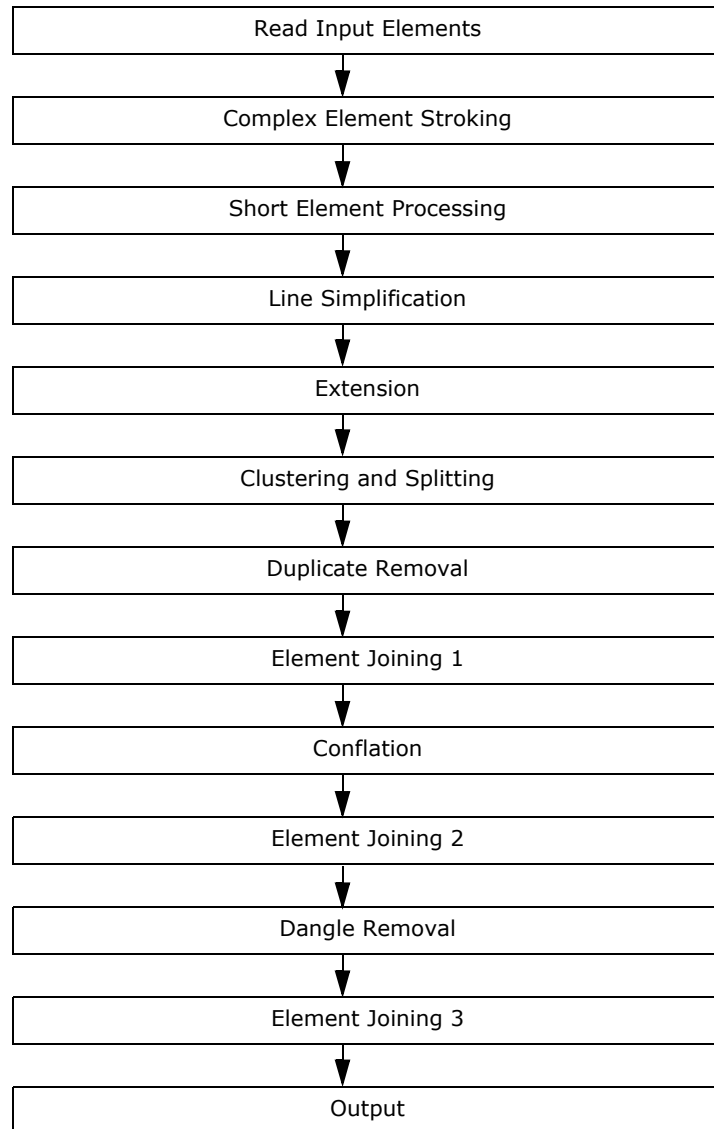
This factory takes features as input and processes them based on the specified modules, tolerance, dangle factor, filter factor, and specific attribute data. This factory is useful for multi-layer and multi-tolerance two-dimensional data cleaning.

The number of layers used in cleaning the data is determined by the number of different tolerance values of input features. Features that have the same tolerances are processed as being on the same layer.

Geometries such as path, polygon, donut, ellipse, elliptical arc, multi-area, multi-curve, text, and multi-text are converted to basic geometries such as point, line, path, arc or multi-point prior to the cleaning process. In other words, the geometries of the features are not necessarily preserved. Input features with invalid geometries are ignored and deleted.

Module Sequence

This default workflow shown below is suitable for most situations. However, using the individual modules, it is possible to create any number of customized workflows for specific projects and/or datasets (for example, in Workbench, by using a series of consecutive MRFCleaner transformers or custom transformers). It is important, however, to understand the data being processed and the desired end result.



General Processing Tips

There are several fundamental tips and tricks for optimizing processing. The following list identifies and briefly describes some key issues to consider when cleaning data.

Know your data

The first step in any cleaning exercise is to become familiar with your source data. Information on data quality (i.e., 1m vs. 100m accuracy), data currency, and intended use is important in determining which cleaning modules and tolerances should be used. If such information is not available, a visual inspection of the design file(s) should provide insight into average line-work gap sizes, line weeding requirements, and other issues which may exist.

Start small

When setting cleaning tolerances, it is always best to start small. With smaller tolerances, the software uses a smaller search radius, which reduces the number of potential element intersections to consider and increases processing speed. Also, if the bulk of the linework errors can be corrected using a small tolerance, more detail can be maintained in the data set. One or more cleaning processes can always be repeated with larger tolerances to increase the number of errors automatically corrected.

Mix it up

Depending on the source dataset, and its intended use, you may achieve better results running the individual modules with different tolerances.

MRFCleanFactory Modules

TOLERANCE

The `TOLERANCE` clause must be specified. This value serves as the default tolerance of the input features if the `TOLERANCE_ATTR` clause is not specified or the features have invalid tolerance value. The minimum value allowed is 0.0.

If the `TOLERANCE_ATTR` is specified and the value of the `<attribute_name>` is greater than or equal to 0.0, this value is used instead of the value specified in `TOLERANCE` clause.

SIMPLIFY

If the `SIMPLIFY` clause is set to “YES”, the line weeding / generalization will be included as part of the cleaning process. This involves the removal of line string

vertices based on a specified tolerance. This process uses a weeding tolerance of the value of $(\text{FILTERFACTOR} * \text{TOLERANCE})$ or $(\text{FILTERFACTOR} * \text{value of TOLERANCE_ATTR})$. The latter is used whenever `TOLERANCE_ATTR` clause is specified and its value is valid. The larger the value of the weeding tolerance, the more vertices will be removed. The default value of `FILTERFACTOR` is 1.0.

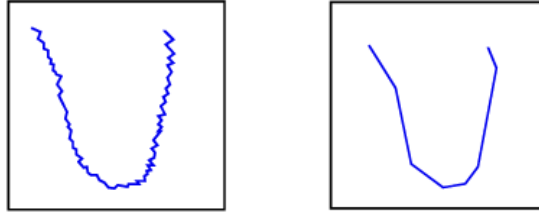


FIGURE 1 Line string before and after simplification

SHORT_ELEMENT

If the `SHORT_ELEMENT` clause is set to “YES”, geometries of features that have lengths smaller than the specified tolerances are deleted. Short geometries created during the cleaning process are also deleted.

EXTEND

If the `EXTEND` clause is set to “YES”, the MRF Extend module is enabled. It is useful to extend certain elements – correcting for undershoot – while maintaining line-work direction. If a feature has a free end, this module will attempt to extend it until it meets other line-work within its tolerance; no intersections are created. This module does not process overshoots; the combination `INTERSECTION` and `DANGLER` modules can be used to serve this purpose.

The `EXTEND` clause processes elements in the following manner:

- line-line extension
- line-arc extension
- arc-arc extension

Line-line extension

- In the figure below, lines AB and CD will be extended to point E, if E is within tolerance of both AB and CD, and both B and D are free ends.

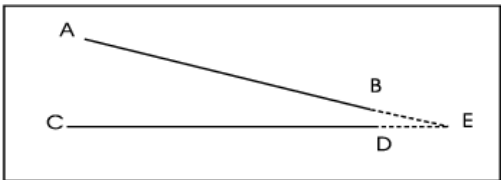


FIGURE 2 Linear extension, example 1

- In Figure 3, lines AB and DE cannot be extended to point F, even though the distance BF is less than the tolerance for AB and EF is less than the tolerance for DE. This is because B is not a free end.

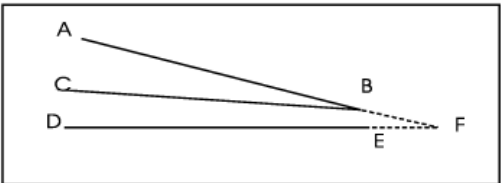


FIGURE 3 Linear extension, example 2

- In Figure 4, CD has a tolerance layer larger than both the distances DE and CD. In this case, point D (rather than point C) will be extended to point E, since it has the smaller extension distance.

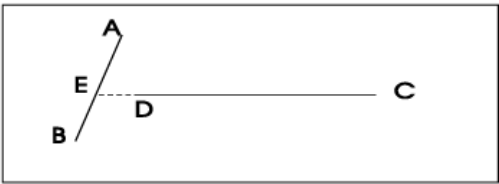


FIGURE 4 Linear extension, example 3

- If a line can be extended to more than one element, it will be extended only as far as the closest one. In Figure 5, line AB will be extended to point C (not D or E).



FIGURE 5 Linear extension, example 4

Line-Arc Extension

Figure 6 shows that circular arc AB will be extended along its path to point C, if BC is less than the tolerance for AB. Also, line DE will be extended to DEF if EF is less than DE's tolerance.

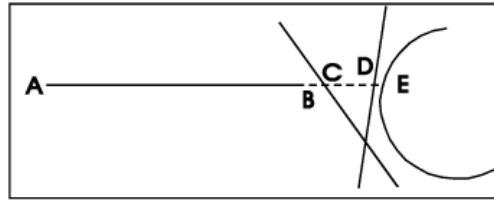


FIGURE 6 Line-arc extension, example 1

Arc-Arc Extension

Figure 7 shows that for arcs AB and DE, if BC is less than the tolerance for AB, then arc AB will be extended to C. Arc DE will be extended to F, if EF is less than the tolerance of DE.

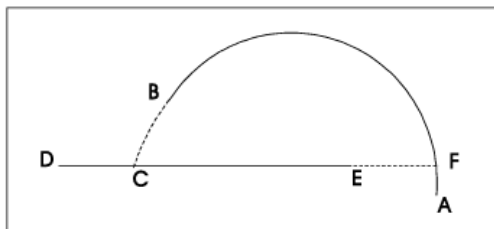


FIGURE 7 Arc-arc extension, example 1

INTERSECT

If the `INTERSECT` clause is set to “YES”, the MRF Intersect module is enabled. This module computes intersections between all input features, breaking arcs and lines wherever an intersection occurs. A fuzzy intersection is also created from geometries which are within one of the tolerance distances, but do not actually touch or cross.

If the `FUZZY_INTERSECT` clause is set to “NO” then fuzzy intersections will not be created in the process.

DUPLICATE_REMOVE

If the `DUPLICATE_REMOVE` clause is set to “YES”, the MRF Duplicate remover module is enabled. Features are considered to be duplicates if their geometries are within tolerance and only features with a smaller tolerance will remain after cleaning.

JOIN

If the `JOIN` clause is set to “YES”, then singly-connected features will be joined to form longer ones. A pair of linear features become candidates for joining only when the two of them are singly connected at a given node or end point.

CONFLATE

If the `CONFLATE` clause is set to “YES”, then the geometry of a feature can be changed match that of another, if the two are approximately the same to begin with.

DANGLER

A dangle is a geometry that has at least one free end point. If the `DANGLER` clause is set to “YES”, MRFCleanFactory will remove dangles if their lengths are less than the $(\text{DANGLEFACTOR} * \text{TOLERANCE})$ or $(\text{DANGLEFACTOR} * \text{value of TOLERANCE_ATTR})$. Again the latter is always used whenever possible. The default value of `DANGLEFACTOR` is 1.0.

See below for a description of the other types of output clauses that are supported.

OBJECT_CLEAN

If the `OBJECT_CLEAN` is set to “YES”, then area features such as polygons or donuts will be cleaned without being stroked first.

Sample Results

The MRFCleanFactory uses generic algorithms to perform data cleaning and does not follow a set of predefined cases. The best way to learn the behavior of MRFCleanFactory is to construct test cases.

The following diagrams illustrate the cleaning principles. Each element is assumed to be on a unique level unless otherwise stated. Tolerances are shown at the top of each figure.

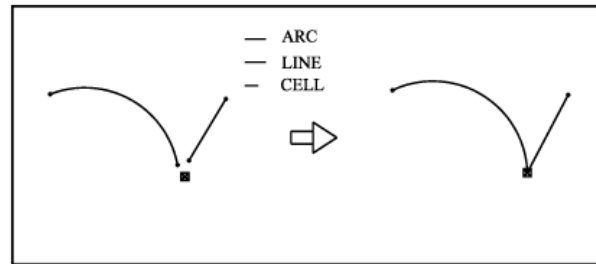
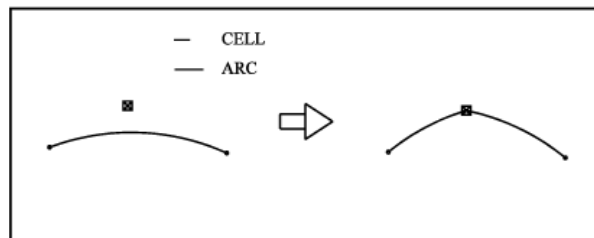


FIGURE 8 Arc and line nodes moved to cell origin



Note: New arcs maintain same radius with different centers.

FIGURE 9 Arc broken and ends collocated to cell origin

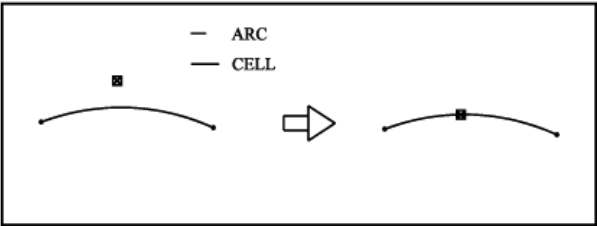


FIGURE 10 Cell moved to arc. Arc is broken.

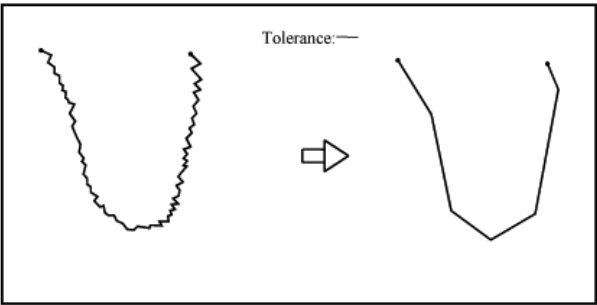


FIGURE 11 Redundant vertices removed by line weeding.

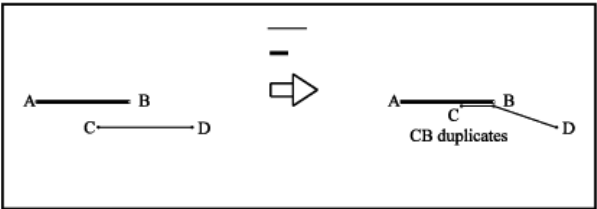


FIGURE 12 Line with larger tolerance moved to line with smaller tolerance

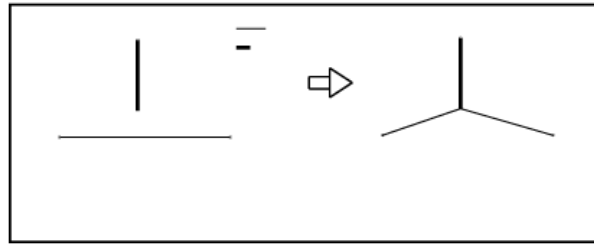


FIGURE 13 Fuzzy intersection created in linear element with larger tolerance

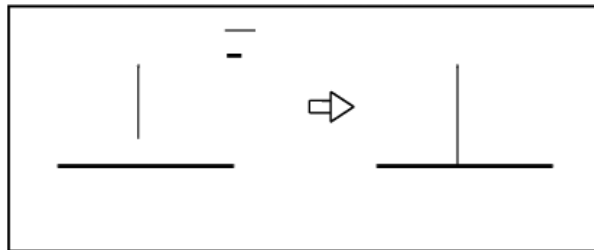


FIGURE 14 Fuzzy intersection created in linear element with smaller tolerance

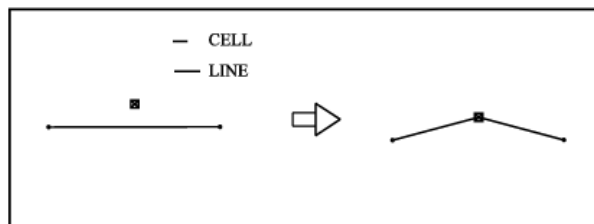


FIGURE 15 Fuzzy intersection created in linear element with larger tolerance

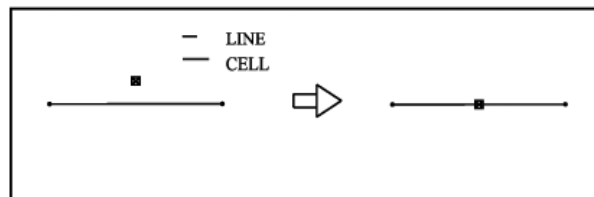


FIGURE 16 Cell with large tolerance moved to fuzzy intersection in linear element

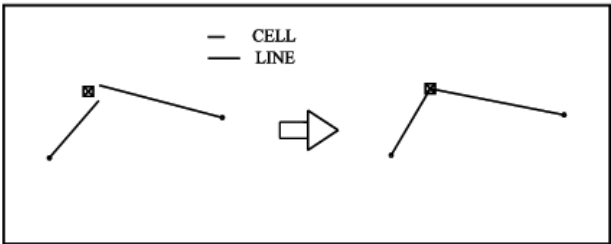


FIGURE 17 Linear elements with large tolerance moved to collocate with cell origin with smaller tolerance

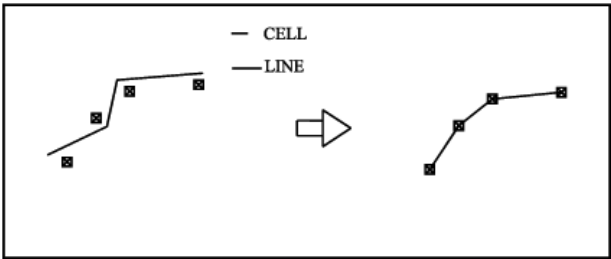


FIGURE 18 Nodes and vertices on linear element with large tolerance collocated at cell origin with smaller tolerances

Assumptions

None.

Output Tags

Tag	Description
CLEANED	This tags output features that are cleaned by the MRFCleanFactory. There is one additional attribute (MRF_CLEAN_STATUS) added on each feature that specifies if the feature is unchanged (“Original”), modified (“Modified”), or created (“Created”) in the process of data cleaning.
FLAGGED	This tags output features that are flagged by the MRFCleanFactory if any of the SHORT_ELEMENT, INTERSECT and Dangler is set to FLAG. There is one additional attribute (MRF_CLEAN_STATUS) added on each feature that specifies if the feature is flagged for being shorter than the tolerance (“short”), dangling (“dangle”), or an intersection point (“intersection”) in the process of data cleaning.

NeighborColorSetterFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```

FACTORY_DEF <ReaderKeyword> NeighborColorSetterFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [ALGORITHM (SIMPLE|FIVE_COLOR)]
  [COLOR_ID_ATTR <color attribute>]
  [SET_COLORS (YES|NO)]
  [AREA_ID_ATTR <area ID attribute>]
  [NEIGHBOR_IDS_ATTR <neighbor IDs attribute>]
  [OUTPUT COLORED
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory assigns colors to areas in a coverage such that adjacent areas are colored differently, and the total number of colors used is kept small.

ALGORITHM

When **ALGORITHM** is set to **SIMPLE**, each area is colored with the first available color. Ideally, only a few colors will be used, but the total number of colors is not guaranteed.

When **ALGORITHM** is set to **FIVE_COLOR**, at most five colors will be used to color all of the regions.

COLOR_ID_ATTR

Colors for each region are output to this attribute as integers (the first color is 0, the second is 1, etc.). If the default fill colors are not desired (see **SET_COLORS** below), this attribute may be manipulated to provide alternate colors for the regions (e.g., mapping from 0 to red, 1 to green, etc.).

SET_COLORS

If **SET_COLORS** is **YES**, pen and fill colors will be assigned to each area based on a default color scheme.

AREA_ID_ATTR and NEIGHBOR_IDS_ATTR

When `AREA_ID_ATTR` and `NEIGHBOR_IDS_ATTR` are specified, these attributes are used to determine which areas are adjacent to each other, and therefore should be colored differently. Each input area must have an ID (a non-negative integer provided in the `AREA_ID_ATTR` attribute) as well as a list of all neighboring areas' IDs (as a comma-delimited list in the `NEIGHBOR_IDS_ATTR` attribute).

When `AREA_ID_ATTR` and `NEIGHBOR_IDS_ATTR` are not specified, adjacencies between areas are determined geometrically. In this case, the input areas must consist solely of polygons that form a coverage (i.e., they must not overlap and there may not be empty space between areas that should be adjacent). Any non-polygon features will be dropped.

Assumptions

None.

Output Tags

This factory supports the following output tags.

Tag	Description
COLORED	The original features, augmented with the <code>COLOR_ID_ATTR</code> attribute.

OracleQueryFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```

FACTORY_DEF <ReaderKeyword> OracleQueryFactory
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [SERVER_TYPE <serverType>]
  [SERVER_NAME <serverName>]
  [USER_NAME <userName>]
  [PASSWORD <password>]
  [DB_NAME <databaseName>]
  [TARGET_LAYER <layerName>]
  [QUERY_SUFFIX <suffix>]
  [WHERE_CLAUSE <whereClause>]
  [INTERACTION <spatialRelationship>]
  [INTERACTION_RESULT <spatialRelationshipTest>]
  [OUTPUT_DUPLICATES]
  [OUTPUT (QUERY|RESULT)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory can be used to retrieve spatial data stored using the Oracle 7 and 8 Spatial Cartridge or *Oracle 8i Spatial in the **relational** mode*. It is capable of performing queries with spatial and/or attribute components. Queries are defined by the features received on the input clause, allowing the FME to perform sophisticated Oracle queries.

Note Use the QueryFactory for Oracle Spatial.

The `OracleQueryFactory` is driven by input features that contain the specifics of the query to be performed. For example, a simple user interface could create such query features. These query features could then be run through the FME either immediately or as a batch job when Oracle database activity from connected users is low.

A query is defined by each input feature received and has the following properties:

- Each feature defines the database instance upon which the query is to be performed enabling the FME to dynamically connect to any Oracle database on the network.
- The geometry of the input feature defines a spatial extent, against which the contents of the Oracle database are compared. **INTERACTION** and **INTERACTION_RESULT** clauses may be specified to use any supported Ora-

cle geometry comparison relationship.

The argument to each the clauses of the `OracleQueryFactory` may be a literal constant, an attribute name preceded by the value-of operator (&), or an attribute value function. If it is an attribute value function, the function will be executed on the query feature and the result will be used as the value for the clause.

For example, suppose the query feature has an attribute named `GEOM_LAYER` whose value is to be the geometric layer from which features are to be extracted. The `OracleQueryFactory` would contain a clause of:

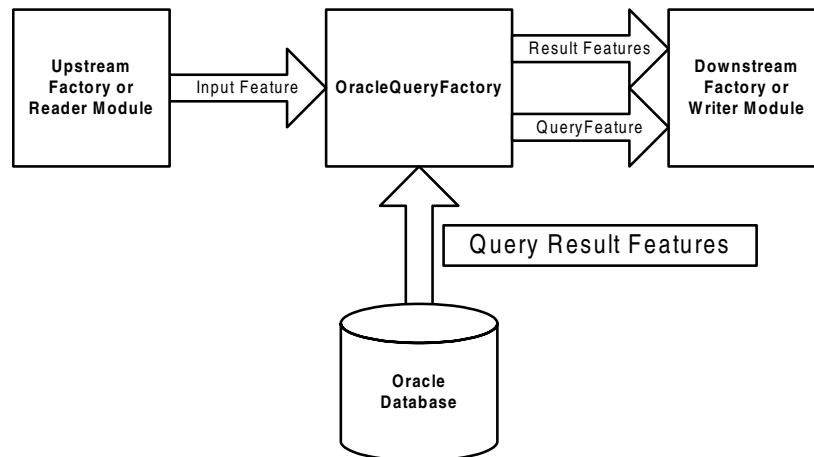
```
TARGET_LAYER &GEOM_LAYER
```

to use this attribute's value as the target layer for the query.

Tip

Features that leave this factory are treated by the FME as if they originated from the reader being used to drive the query.

The figure below illustrates how the `OracleQueryFactory` operates:



OracleQueryFactory Processing

A feature flows into the `OracleQueryFactory` either from the reader module or from an upstream factory. The features the `OracleQueryFactory` accepts are called *query features* as they contain the parameters for the query to be performed. The `OracleQueryFactory` uses these parameters to issue a query against the Oracle database.

As features are returned from the database, they are passed out of the `OracleQueryFactory` without delay for immediate processing by the rest of the system.

After the last feature for a query has been returned from the database, the `OracleQueryFactory` then passes the *query feature* to the rest of the FME system via the `OUTPUT QUERY` clause. The next query is started when the next query feature arrives.

Assumptions

Oracle 7, 8, or 8i is installed on the database server from which you wish to retrieve data.

To perform spatial queries, the target feature layer must have a complete spatial index. When Oracle’s relational method of storing spatial data is used, the `<layerName>_SDOINDEX` table must have been populated using Oracle’s `SDO_ADMIN.POPULATE_INDEX` and/or `SDO_ADMIN.UPDATE_INDEX` utilities. It is also highly recommended that database indexes be created on `<layerName>_SDOGEOM`’s `SDO_GID` column, and on `<layerName>_SDOINDEX`’s `SDO_CODE` and `SDO_GID` columns.

Clauses

The following clauses are used to configure the *OracleQueryFactory* for a retrieval operation:

Clauses	Description	Optional
SERVER_NAME	The name of the service used to perform the query operation.	No
SERVER_TYPE	The type of database used to perform the query operation. Typically the value is ORACLE.	No
DB_NAME	The name of the database upon which the query is performed. This is not normally required for an Oracle database.	Yes
USER_NAME	The <code>userName</code> used to perform the query.	No
PASSWORD	The user’s password.	No
TARGET_LAYER	This establishes the layers (tables) which are targeted by the query. The geometry on each of the specified layers will be compared to the geometry of the field of the search feature, to form the result features. If more than one layer is to be specified in the field, then the layer names must be separated by colons.	No

Clauses	Description	Optional
QUERY_SUFFIX	When performing spatial queries, the query features are written out to a Spatial Cartridge layer which is separate from the features being queried. The name of this query layer is generated by appending a suffix to the name of the layer being queried (for example, <code>ROADS_QUERY</code>). This clause specifies the part that is appended to the geometry layer name to form the query layer name. (In the above example, the query suffix would be specified as “ <code>_QUERY</code> ”.)	No
WHERE_CLAUSE	This specifies an SQL WHERE clause, which is applied to the table’s columns to limit the resulting features. This feature is currently limited to apply only to the attributes of the target Spatial Cartridge layer, and does not allow for joining multiple tables together. By default, there is no WHERE clause applied to the results.	Yes
INTERACTION	This specifies the type of relationship which must exist between the query feature and the geometry in the target layer. Any supported relationship, or combination of relationships, may be specified. Default: <code>ANYINTERACTION</code>	Yes
INTERACTION_RESULT	This specifies the test that is applied to the results of the above geometry relationship comparison. For combined relationships, one might want to use a comparison such as “ <code>= 'TOUCH'</code> ” instead of the default. Default: <code><> 'FALSE'</code>	Yes
OUTPUT_DUPLICATES	If specified, the factory will not ensure that each spatial feature is output only once. The default behaviour for the factory is to keep track of the spatial features. In the event that a single factory performs multiple queries, it ensures that each spatial feature is only output once. Specifying <code>OUTPUT_DUPLICATES</code> removes this check and results in duplicates being output.	Yes

Specifying Spatial Relationships

The set of features returned by the `OracleQueryFactory` is constrained to the set of features which satisfy some spatial interaction with the query feature.

The interaction is specified by providing both an interaction relationship and a result of testing that relationship. Except in the case of `ANYINTERACTION`, the name for type of relationship is returned for a positive match, rather than a value of `TRUE`. For this reason, the default test for the interaction result is “<> ‘FALSE’” rather than (the perhaps more intuitive) “= ‘TRUE’”.

Tip

Currently the `OracleQueryFactory` uses the bounding box of the query feature, rather than the whole feature boundary, to perform spatial queries. A future release may take the query feature’s complete boundary and interior into account.

This table lists the valid geometry interaction relationships.

Search Method	Description
DISJOINT	Returns <code>DISJOINT</code> if the objects have no common boundary or interior points; otherwise, returns <code>FALSE</code> .
EQUAL	Returns <code>EQUAL</code> if the objects share every point of their boundaries and interior, including any holes in the objects; otherwise, returns <code>FALSE</code> .
INSIDE	Returns <code>INSIDE</code> if the geometry feature is entirely contained within the query feature; otherwise, returns <code>FALSE</code> .
OVERLAPBDYDISJOINT	Returns <code>OVERLAPBDYDISJOINT</code> if the objects overlap, but their boundaries do not interact; otherwise, returns <code>FALSE</code> .
OVERLAPBDYINTERSECT	Returns <code>OVERLAPBDYINTERSECT</code> if the objects overlap, and their boundaries intersect in one or more places; otherwise, returns <code>FALSE</code> .
TOUCH	Returns <code>TOUCH</code> if the two objects share a common boundary point, but no interior points; otherwise, returns <code>FALSE</code> .
ANYINTERACTION	Returns <code>TRUE</code> if the objects interact according to any of the above relationships; otherwise, returns <code>FALSE</code> .

In addition to specifying a single relationship, you may specify a combination of relationships to be tested by concatenating them with a plus sign (“+”). For example, if the `INTERACTION` clause is specified as “`INSIDE + TOUCH`”, the result of the interaction test will be one of: “`INSIDE`”, “`TOUCH`” or “`FALSE`”.

Output Tags

The `OracleQueryFactory` supports the following output tags.

Tag	Description
QUERY	The query feature that entered the factory.
RESULT	Each of the features that satisfied the query.

Example

The following example shows the `OracleQueryFactory` being used to retrieve features. Notice the use of macros for specifying the database connection information.

To clarify the use of attribute values as values for the clauses, the features have a target layer and where clause assigned to attributes on the `INPUT` line, and are referenced by value in the clause definitions. These could more easily have been specified as literal strings in the clause definitions.

```

FACTORY_DEF * OracleQueryFactory                                \
  INPUT FEATURE_TYPE queryFeature                               \
    @SupplyAttributes(GEOM_LAYER, roads)                        \
    @SupplyAttributes(WHERE_CLAUSE, "NUM_LANES = 2")           \
  SERVER_NAME $(serverName)                                     \
  SERVER_TYPE $(serverType)                                     \
  USER_NAME $(userName)                                         \
  PASSWORD $(password)                                           \
  TARGET_LAYER &GEOM_LAYER                                       \
  WHERE_CLAUSE &WHERE_CLAUSE                                     \
  INTERACTION "TOUCH + INSIDE"                                   \
  OUTPUT RESULT FEATURE_TYPE *
```


OverlayFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```

FACTORY_DEF <ReaderKeyword> OverlayFactory
  [FACTORY_NAME <factory name>]
  [INPUT (POLYGON|POINT|LINE)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]+
  [LIST_NAME <list name>]
  [OVERLAP_COUNT_ATTRIBUTE <attribute name>]
  [SEPARATE_COLLINEAR_SEGMENTS [(YES|NO)]]
  [TOLERANCE <distance>]
  [GROUP_BY [<attribute name>]+]*
  [OUTPUT (POLYGON|POINT|LINE|ILLEGAL_GEOM)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory performs an overlay operation. During the overlay, all intersections and overlaps of the input features are computed, and the attributes of features that interact are merged. Merging adds new attributes to a feature but it will not replace or alter existing attributes. If necessary, new features will be created for each area or line segment created as a result of the overlay.

When new points are added during the intersection calculations, if the original lines had measures, the new points will have the same measures, interpolated between the nearest points by distance along the connecting segment. For adding points to arcs, the distance around the perimeter of the arc is used.

This factory has six distinct modes of operation:

- Mode 1: Polygon Overlay
- Mode 2: Point on Polygon Overlay
- Mode 3: Line on Polygon Overlay
- Mode 4: Point on Line Overlay
- Mode 5: Point Overlay
- Mode 6: Line Overlay

The mode of the factory is determined by which combination of input tags are present.

Any aggregates input to the factory are split up and treated as though they were separate features.

If the `FME_GEOMETRY_HANDLING` directive is set to yes in the mapping file, arcs are regarded as linear features, and ellipses are regarded as polygonal features; otherwise, arcs are regarded as point features, and ellipses can be regarded as either point or stroked polygonal features (depending on their input tag).

Mode 1: Polygon Overlay

This mode is entered only when the `POLYGON` input tag is present in the factory definition.

In this mode, input polygons are analyzed and new polygons are formed for each area that results when the original polygons are overlaid on top of each other. The resulting polygons have all their attributes merged. If the `LIST_NAME` clause is present, each output area will have an attribute list containing the attributes from each area it overlaps. As well, if the `OVERLAP_COUNT_ATTRIBUTE` clause is present, an attribute will be added indicating the number of input features the area overlaps.

In this mode, only the `POLYGON` output tag is used. If any other output tags are specified, they will cause an error during factory creation.

Mode 2: Point on Polygon Overlay

This mode is entered when the `POLYGON` and `POINT` input tags are present in the factory definition.

In this mode, each input point is tested against each input polygon. If the polygon contains the point, the attributes of the point are merged with the polygon and the attributes of the polygon are merged with the point. If the `LIST_NAME` clause is present, the attributes of each polygon containing the point are added to the point's list, and the attributes of each point contained by a polygon are added to the polygon's list. As well, if the `OVERLAP_COUNT_ATTRIBUTE` clause is present, an attribute will be added to each polygon indicating the number of points it contained, and similarly each point will have an attribute which holds the number of polygons it was inside.

Points that lie on the boundary of a polygon are considered to be contained in the polygon.

In this mode, the `POLYGON` and `POINT` output tags are available to be used.

Mode 3: Line on Polygon Overlay

This mode is entered when the `POLYGON` and `LINE` input tags are present in the factory definition.

In this mode, each line is segmented at the polygon boundaries. Then each resulting line is checked against all the original polygons. If the polygon

contains the resultant line, the attributes of the line are merged with the polygon and the attributes of the polygon are merged with the line. If the `LIST_NAME` clause is present, the attributes of each polygon containing the line are added to the line's list, and the attributes of each line contained by a polygon are added to the polygon's list. As well, if the `OVERLAP_COUNT_ATTRIBUTE` clause is present, an attribute will be added to each polygon indicating the number of lines it contained, and similarly each line will have an attribute that holds the number of polygons it was inside.

Lines that lie on the boundary of a polygon (collinear) are considered to be contained in the polygon.

In this mode, the `POLYGON` and `LINE` output tags are available to be used.

Mode 4: Point on Line Overlay

This mode is entered when the `POINT` and `LINE` input tags are present in the factory definition.

In this mode, each line is segmented at the points if on the line or at the closest coordinates to the points. Then each resulting line is checked against all the points. If the `TOLERANCE` clause is present, the value is compared to the distance from the lines to the points and the lines will be segmented if the distance is less than or equal to the `TOLERANCE` value. When such a match occurs, the attributes of the segmented lines are merged with the points and the attributes of the points are merged with the lines. If the `LIST_NAME` clause is present, the attributes of each point used to segment the line are added to the line's list, and the attributes of each line that the point segmented are added to the point's list. As well, if the `OVERLAP_COUNT_ATTRIBUTE` clause is present, an attribute will be added to each line indicating the number of points it was segmented by, and similarly each point will have an attribute that holds the number of lines it segmented.

In this mode, the `POINT` and `LINE` output tags are available to be used.

Mode 5: Point Overlay

This mode is entered when only the `POINT` input tag is present in the factory definition.

In this mode, input points within the `TOLERANCE` distance specified are matched up with each other. Each output point has the attributes of all other points within the tolerance merged onto it. If the `LIST_NAME` clause is present, each output point will also have an attribute list containing the attributes from each point which is within the `TOLERANCE` distance. As well, if the `OVERLAP_COUNT_ATTRIBUTE` clause is present, an attribute will be added indicating the number of nearby points. This mode provides similar functionality to the `ProximityFactory`.

In this mode, only the `POINT` output tag is used. If any other output tags are specified, they will cause an error message during the translation.

Mode 6: Line Overlay

This mode is entered when only the `LINE` input tag is present in the factory definition.

In this mode, input lines are intersected and matched up with each other. In places where the lines were collinear, one line segment is output with all the original feature attributes merged. Points are also created at the locations where lines touched, again with all the original feature attributes merged. If the `LIST_NAME` clause is present, each output *POINT* will have an attribute list containing the attributes from each line that touched it. As well, if the `OVERLAP_COUNT_ATTRIBUTE` clause is present, an attribute will be added indicating the number of lines touching the output feature..

If the `SEPARATE_COLLINEAR_SEGMENTS` clause is specified, any line segments which are collinear are not merged together; instead, one copy is output for each original feature which shared that segment.

If the `SEPARATE_COLLINEAR_SEGMENTS` clause is not specified, any line segments which are collinear are merged. If the `LIST_NAME` clause is present, all output lines will have an attribute list containing the attributes from each collinear line that was merged. If the `SEPARATE_COLLINEAR_SEGMENTS` clause is specified, then the output lines will NOT have an attribute list, regardless of the `LIST_NAME` clause.

In this mode, the `POINT` and `LINE` output tags are available to be used. If any other output tags are specified, they will cause an error message during the translation.

Assumptions

This factory makes no assumptions about the input data. In particular, the polygons that are input to the factory may overlap.

Input Tags

The `OverlayFactory` supports the following input tags.

Tag	Description
LINE	Linear features to be overlayed in a fashion determined by the combination of input tags, as specified above.
POINT	Point features to be overlayed in a fashion determined by the combination of input tags, as specified above.

Tag	Description
POLYGON	Area features to be overlayed in a fashion determined by the combination of input tags, as specified above.

Output Tags

The OverlayFactory supports the following output tags.

Tag	Description
LINE	Any linear features input to or created by the overlay operation.
POINT	Any point features input to or created by the overlay operation.
POLYGON	Any area features input to or created by the overlay operation.
ILLEGAL_GEOM	Any features that were input with a geometry type not matching the input tag (for instance, a line input with the POLYGON tag.)

Example 1

In this example, the OverlayFactory is used to overlay `owners` polygons with `soilType` polygons. The resultant polygons contain the attributes of each of the polygons they overlap. If the `owners` and `soilType` input polygons each formed a non-overlapping coverage with precisely the same spatial extent, then the `numOverlaps` attribute should have the value 2 on each output polygon.

```
FACTORY_DEF * OverlayFactory \
    INPUT POLYGON FEATURE_TYPE owners \
    INPUT POLYGON FEATURE_TYPE soilType \
    LIST_NAME attrList \
    OVERLAP_COUNT_ATTRIBUTE numOverlaps \
    OUTPUT POLYGON FEATURE_TYPE ownerSoil
```

Example 2

In this example, the OverlayFactory is used to overlay `owners` polygons with `road` lines. The resultant `road` line features contain the attributes of each of the polygons they overlap, and the original `owners` polygons are output with the attributes of any `road` features they contain added to them. If the `owners` input polygons formed a non-overlapping coverage with the same spatial extent as the `road` lines, then the `numOverlaps` attribute found on each output line should have the value 1, because each line would be in exactly one polygon.

```
FACTORY_DEF * OverlayFactory \
    INPUT POLYGON FEATURE_TYPE owners \
    INPUT LINE FEATURE_TYPE road \
    LIST_NAME attrList \
```

```

OVERLAP_COUNT_ATTRIBUTE numOverlaps \
OUTPUT POLYGON FEATURE_TYPE owners \
OUTPUT LINE FEATURE_TYPE roadPieces

```

Example 3

In this example, the `OverlayFactory` is used to overlay `owners` polygons with `well` points. When output, the `well` point features will contain the attributes of each of the polygons they are contained by, and the original `owners` polygons will be output with the attributes of any `well` features they contain added to them. If the `owners` input polygons formed a non-overlapping coverage with the same spatial extent as the `well` points, then the `numOverlaps` attribute found on each output point should have the value 1, because each point would be in exactly one polygon.

```

FACTORY_DEF * OverlayFactory \
INPUT POLYGON FEATURE_TYPE owners \
INPUT POINT FEATURE_TYPE wells \
LIST_NAME attrList \
OVERLAP_COUNT_ATTRIBUTE numOverlaps \
OUTPUT POLYGON FEATURE_TYPE owners \
OUTPUT POINT FEATURE_TYPE wells

```

```
FACTORY_DEF <ReaderKeyword> PIPComponentsFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [OUTPUT (POINT|POLYGON) FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
```

Tag	Description
POINT	The point portion of the input feature's geometry is output as a point feature. It has all attribute values of the input feature.
POLYGON	The polygon or donut polygon that defined the input PointInPolygon feature is output with this clause. The output polygon has all attributes of the input feature.

Example

The following example accepts `ForestCoverPIP` features types and outputs two types of features: `InteriorPoint` and `ForestCoverPolygon`. Each of the output features is assigned all attributes of the original feature.

```
FACTORY_DEF IGDS PIPComponentsFactory \
  INPUT FEATURE_TYPE ForestCoverPIP \
  OUTPUT POINT FEATURE_TYPE InteriorPoint \
  OUTPUT POLYGON FEATURE_TYPE ForestCoverPolygon
```

The second example shows how this factory can be used with the `@GeneratePoint` feature function to replace a polygonal feature with a feature that has all the same attributes, but only the geometry of a point internal to the original polygon:

```
FACTORY_DEF Shape PIPComponentsFactory \
  INPUT FEATURE_TYPE tract @GeneratePoint() \
  OUTPUT POINT FEATURE_TYPE tractPoint
```


PolygonDissolveFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> PolygonDissolveFactory
[FACTORY_NAME <factory name>]
[INPUT_FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*
[GROUP_BY [<attribute name>+]*]
[LIST_NAME <list name>]
[DISSOLVE_COUNT_ATTRIBUTE <attribute name>]
[SUM_FIELDS [<attribute name>+]]
[AVERAGE_FIELDS [<attribute name>+]]
[WEIGHTED_AVERAGE_FIELDS [<attribute name>+]]
[NON_OVERLAPPING_INPUT [(yes|no)]]
[OUTPUT (POLYGON|INTERIOR_LINE|NON_POLYGON)
  FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*

```

This factory accepts polygonal features and outputs dissolved polygons. Dissolved polygons are those polygons formed when shared edges between adjacent polygons are removed. The factory assumes the input polygons are properly noded. Any features that are not polygons are output untouched via the *NON_POLYGON* feature output tag.

The input polygonal features may be partitioned into groups for dissolving by using the **GROUP_BY** clause. If the **GROUP_BY** clause is not specified, then all input features are processed together. The **GROUP_BY** clause enables a single factory to perform the dissolve of several different, potentially overlapping, polygonal coverages.

By default, no attributes other than the **GROUP_BY** attributes are carried across from the input features to the output features. However, if any of the **LIST_NAME**, **DISSOLVE_COUNT_ATTRIBUTE**, **AVERAGE_FIELDS**, **SUM_FIELDS**, or **WEIGHTED_AVERAGE_FIELDS** clauses are present, then the output features will have the attributes of their constituent input features merged into them. In this case, the attributes of the largest constituent feature are copied to the output feature, and any different attributes found in any other constituent features are also added.

The **LIST_NAME** clause identifies a list into which the attributes of the constituent features are stored. The largest constituent feature's attributes are the first elements in the list, but no order is defined for the remaining elements.

The number of polygons dissolved into an output polygon is stored in the attribute identified by the **DISSOLVE_COUNT_ATTRIBUTE** clause.

The `SUM_FIELDS` clause identifies attributes whose values are to be summed together when constituent polygons are dissolved and then assigned to the output polygons.

The `AVERAGE_FIELDS` clause identifies attributes whose values are averaged during the dissolve. The `WEIGHTED_AVERAGE_FIELDS` clause identifies attributes whose values are averaged during the dissolve weighted according to the area of their original polygon.

If the `NON_OVERLAPPING_INPUT` clause is given with an argument of `no`, then for any two areas to be dissolved together, they **MUST** share a common boundary with identical vertices. For example, a small area entirely contained inside another area without sharing any edges will **NOT** be dissolved with the containing area. (Note that this keyword has non-standard behavior within FME: if it is not given in the factory definition, then its default value is `yes`.)

If the `OUTPUT INTERIOR_LINE` clause is specified, then any lines that are not part of the output polygons will be output. This requires substantial additional computation and should only be specified if these lines are truly required.

Additional statistical information about the operation of the `PolygonDissolveFactory` may be output by specifying `FME_DEBUG PolygonDissolveFactory` in the mapping file. Search *Mapping File Debugging* in the *FME Fundamentals* manual, available in PDF format in the Download Centre at www.safe.com.

If the `FME_GEOMETRY_HANDLING` directive is set to “yes” in the mapping file, ellipses are accepted as polygonal features; they are otherwise output untouched via the `NON_POLYGON` feature output tag.

Assumptions

This factory assumes that all input polygons are already topologically noded; that is, a vertex is present at each intersection point. This factory also assumes that the polygons do not overlap each other.

If these assumptions are not held, false results will be produced.

Output Tags

The `PolygonDissolveFactory` supports the following output tags.

Tag	Description
POLYGON	Dissolved Polygon features with attributes specified by <code>GROUP_BY</code> , <code>SUM</code> , and <code>MEAN</code> clauses set appropriately.

NON_POLYGON	Non-polygonal features that entered the factory exit the factory here untouched.
INTERIOR_LINE	Linear features that represent the portions of the input polygons which are not part of the output dissolved polygon.

Example 1

The following example takes all input features from an input Shape file and passes all non-polygon features on via its NON_POLYGON output tag. All input polygons are grouped by their `long_id`, `category`, and `full_id` attribute values. All output polygons have their feature type set to the value of the `category`. The factory also outputs any linear features that represent the shared edges between the dissolved polygons. These shared edges are output via the `INTERIOR_LINE`.

```

FACTORY_DEF SHAPE_IN PolygonDissolveFactory \
  INPUT FEATURE_TYPE * \
  GROUP_BY long_id category full_id \
  OUTPUT POLYGON FEATURE_TYPE * @FeatureType(&category) \
  OUTPUT INTERIOR_LINE FEATURE_TYPE InteriorLine \
  OUTPUT NON_POLYGON FEATURE_TYPE *

```

Example 2

In this example, attribute values are calculated during the polygon dissolve. During input to the factory, the area of the feature is calculated and stored as an attribute. In addition, the `TreeCoverage` attribute value is copied into a new attribute called `WeightedTreeCoverage` using the `@SupplyAttributes` function.

```

FACTORY_DEF * PolygonDissolveFactory \
  INPUT FEATURE_TYPE * \
  Area @Area() \
  @SupplyAttributes(WeightedTreeCoverage,&TreeCoverage) \
  LIST_NAME constituents \
  DISSOLVE_COUNT_ATTRIBUTE dissolveCount \
  SUM_FIELDS Population Area \
  AVERAGE_FIELDS TreeCoverage \
  WEIGHTED_AVERAGE_FIELDS WeightedTreeCoverage \
  OUTPUT POLYGON FEATURE_TYPE pops

```

If these features are input to the factory:



Feature Type: pops	
Attribute Name	Value
Area	388959.852293596
Population	100000
TreeCoverage	50
WeightedTreeCoverage	50
fme_geometry	fme_polygon
Coordinates: (499442.4775,500371.6815) (499442.4775,499862.832) (500132.7435,499785.398) (500141.593,500396.0175) (499442.4775,500371.6815)	



Feature Type: pops	
Attribute Name	Value
Area	145841.465747103
Population	50000
TreeCoverage	10
WeightedTreeCoverage	10
fme_geometry	fme_polygon
Coordinates: (500141.593,500396.0175) (500619.469,500409.292) (500132.7435,499785.398) (500141.593,500396.0175)	

then this feature will be output:



Feature Type: pops	
Attribute Name	Value
Area	534801.318040699
Population	150000
TreeCoverage	30
WeightedTreeCoverage	39.0919142621877
constituents{0}	Area is 388959.852293596'
constituents{0}	Population is 100000
constituents{0}	TreeCoverage is 50
constituents{0}	WeightedTreeCoverage is 50
constituents{1}	Area is 145841.465747103
constituents{1}	Population is 50000
constituents{1}	TreeCoverage is 10
constituents{1}	WeightedTreeCoverage is 10
dissolveCount	2
Coordinates: (500132.7435,499785.398) (500619.469,500409.292) (500141.593,500396.0175) (499442.4775,500371.6815) (499442.4775,499862.832) (500132.7435,499785.398)	

PolygonFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> PolygonFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [REMOVE_CORRIDORS]\
  [PRESERVE_ORIENTATION (CLOCKWISE|COUNTER_CLOCKWISE)]
  [FLUSH_BEFORE_CURRENT_WHEN <value> <operator> <value>]
  [FLUSH_AFTER_CURRENT_WHEN <value> <operator> <value>]
  [GROUP_BY [<attribute name>+]*]
  (END_NODDED|VERTEX_NODDED)
  [REPORT_PROGRESS [(yes|no)]]
  [ALLOW_CYCLES [(yes|no)]]
  [LIST_NAME <list attribute name>{}]
  [OUTPUT (POLYGON|LINE)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory takes linear features to form topologically correct polygons. Any lines that cannot be formed into polygons are joined together to create maximum length linestrings. The input linear features may be partitioned by the **GROUP_BY** clause. If the **GROUP_BY** clause is not specified, then all inputs will be processed together. The output features have the same coordinate system as the partitioned input features. If there are conflicting coordinate systems set (empty coordinate systems are considered non-conflicting) within a partition, then the output feature will have no coordinate system set. No attributes are carried across from the **INPUT** features to the output features. However, all **OUTPUT** features are assigned the **GROUP_BY** attributes before being output. If the **OUTPUT LINE** clause is not specified, then lines that are not part of a polygon will be deleted.

The **REMOVE_CORRIDORS** directive is used to preprocess any input linework to remove any zero-width corridors that may have been present. Such corridors are used by some systems to connect holes to their enclosing polygon. This clause has no use unless the data comes from this type of a system. When it is used, **VERTEX_NODDED** must be specified.

The **END_NODDED** and **VERTEX_NODDED** directives tell the factory about the topology of the input features. **END_NODDED** indicates that all features begin and end at topologically significant points, and that none of their vertices connect with any other features. **VERTEX_NODDED** indicates that every vertex may be topologically significant and must be considered when looking to form polygons. **Note:** If these settings do not accurately represent the input data, you

may get undesirable results (including dropped features and missing data). Ensure that the lines are topologically correct. They must not self-intersect or intersect each other, and they must close at nodes.

ALLOW_CYCLES specifies that coordinate “cycles” within a polygon are allowable; such polygons might be considered invalid by other parts of FME or by output formats. A “cycle” is a line segment that occurs twice in the same polygon's boundary (once in each direction).

If **LIST_NAME** is given, then a list will be created on each output feature, containing an element for each input feature which contributed to that geometry, in order of appearance.

The **REPORT_PROGRESS** directive tells the factory to log statistics about the factory's progress during translation.

The **PRESERVE_ORIENTATION** directive instructs the factory to follow the arc directions. This is useful when building polygons from a series of lines that do not form a polygonal coverage and polygons are to be constructed only where the directed arcs form polygons. If **CLOCKWISE** is specified, then only polygons with arcs directed in a clockwise direction will be built. If **COUNTER_CLOCKWISE** is specified, then only polygons with arcs directed in a counterclockwise direction will be built.

If the **FLUSH_BEFORE_CURRENT_WHEN** tag is specified, the factory will perform the test defined by this clause every time it receives an input feature. If the result of the test is true, then the factory will flush out all stored features via the **OUTPUT** clauses before processing the input feature. If the result is false, the feature will be processed and the factory will not be flushed. The **FLUSH_AFTER_CURRENT_WHEN** tag operates identically except that the factory is flushed **after** the current input feature processed instead of before.

The <operator> is one of <, >, =, !=, >=, <=.

The <value> may be a literal constant, an attribute name preceded by the value-of operator (&), or an attribute value function. If it is an attribute value function, the function will be executed on the current feature and the result will be used for the test.

The example **FLUSH_BEFORE_CURRENT_WHEN** and **FLUSH_AFTER_CURRENT_WHEN** clauses include:

```
FLUSH_BEFORE_CURRENT_WHEN @Area() < 100
FLUSH_AFTER_CURRENT_WHEN &numLanes > 2
FLUSH_BEFORE_CURRENT_WHEN "Joe" = "Jerry"
```

At run-time, the PolygonFactory decides to invoke numeric comparisons or string comparisons, as greater than and less than, that have different meanings depending on the type of operands. If both arguments may be converted to

numbers, then numeric comparisons will be used, otherwise string comparisons will be used.

The factory processes all input features and outputs appropriate results each time it is flushed, or once the last feature is input to the factory.

If the `FME_GEOMETRY_HANDLING` directive is set to yes in the mapping file, arcs and ellipses are accepted as linear features; they are otherwise rejected as point features.

Assumptions

This factory assumes that all input lines are topologically noded.

Output Tags

The `PolygonFactory` supports the following output tags.

Tag	Description
LINE	Maximum length of line features that are not part of any polygon.
POLYGON	Polygon features formed from the input linework.

Example

The following example takes Design file lines from level 9 and forms them into polygons. The polygons are output as features of the type `ForestCoverPolygon`. Any lines that cannot be formed into polygons are output as features of the type `PolygonFactory_Line`.

```

FACTORY_DEF IGDS PolygonFactory                                \
  INPUT FEATURE_TYPE 9 igds_type igds_line                    \
  END_NODED                                                    \
  OUTPUT LINE FEATURE_TYPE PolygonFactory_Line                \
                                                                \
                                                                lineNum @Count (LineNum) \
  OUTPUT POLYGON FEATURE_TYPE ForestCoverPolygon              \
                                                                \
                                                                polyNum @Count (PolyNum)

```

TIP Notice the use of the `@Count()` function on the output statements to assign a unique number to each `PolygonFactory_Line` and `ForestCoverPolygon` output feature.

ProximityFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```

FACTORY_DEF <ReaderKeyword> ProximityFactory
  [FACTORY_NAME <factory name>]
  [INPUT (BASE|CANDIDATE) FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  SPATIAL_MATCH (CLOSEST <maxDistance>|ENVELOPE_INTERSECT)
  [TEST <value> <operator> <value>]+
  [TESTED_CANDIDATES_LIST <list attribute name>]
  [CLOSE_CANDIDATES_LIST <list attribute name>]
  [GROUP_BY [<attribute name>]+]*
  [OUTPUT (MATCHED|UNMATCHED_BASE|UNMATCHED_CANDIDATE)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory performs feature matching based on the spatial relationship between features.

When CLOSEST <maxDistance> is specified:

- The factory first stores all BASE features in one set and all CANDIDATE features in another set until it receives all features.
- After all features have been received, the factory then looks at each BASE feature and finds the closest CANDIDATE feature within <maxDistance> distance of the BASE feature. If no CANDIDATE feature is found to be within <maxDistance> distance, then the BASE feature will be output via the UNMATCHED_BASE output tag. The units of <maxDistance> are those of the input features.
- If a CANDIDATE feature is found, then the factory will add all attributes from the closest CANDIDATE feature to the BASE feature and will send the BASE feature out of the factory via the MATCHED tag. It will also add the distance (in ground units) from the BASE to the CANDIDATE in the attribute distance. If the same attribute exists on the BASE and the CANDIDATE, the BASE feature's attribute value will be overwritten by the value contained in the CANDIDATE's attribute. The coordinates of the closest BASE point and the closest CANDIDATE point are added to the matched BASE feature to the corresponding attributes: closest_base_x, closest_base_y, closest_candidate_x, and closest_candidate_y. The angle between the closest BASE point and the closest CANDIDATE point is also added to the BASE feature on the attribute angle. Finally, the angle from the closest candidate point to its immediate successor within the candidate feature is stored in the attribute candidate_angle. (If the closest candidate point is

the last point in the feature, then `candidate_angle` will contain the angle from the previous point of the candidate feature to the closest candidate point.) Also, it adds an attribute named `candidate_label_angle` which contains the `candidate_angle` adjusted such that if a label is created with that angle, it will read from left to right.

- Once all `BASE` features have been processed, the factory outputs any `CANDIDATE` features that have not been matched via the `UNMATCHED_CANDIDATE` output tag.
- It is important to note that if a `CANDIDATE` feature is the closest feature to multiple `BASE` features, then it will be combined with multiple `BASE` features.

Note The `BASE` input clause is optional. If only `CANDIDATES` are supplied to the factory, then they are also used as the `BASE`s. When considering each feature as a `BASE`, the feature itself will be ignored as a `CANDIDATE`; i.e., every feature will not find itself as the closest candidate.

When `ENVELOPE_INTERSECT` is specified, the following is performed:

- The `CANDIDATE` features are held until the first `BASE` feature is encountered. All `BASE` features other than the first are output via the `UNMATCHED_BASE` clause.

The `TEST` clause here specifies the comparison done between the `BASE` and `CANDIDATE` features. To refer to `BASE` attributes, prefix the attributes with “`BASE.`” and to refer to `CANDIDATE` attributes, use the prefix “`CANDIDATE.`”. Note that just prior to the test being performed, the `BASE` features have the `closest_base_x`, `closest_base_y`, `closest_candidate_x`, and `closest_candidate_y` attributes added. Furthermore, an unprefix angle and distance attribute is available to be used in the test as well.

The following example specifies a test which will only allow `BASE` and `CANDIDATE` features to be matched when they have the same value in their `ID` attribute:

```
TEST &BASE.ID = &CANDIDATE.ID
```

When the `TESTED_CANDIDATES_LIST` clause is specified, an attribute list with the specified name is created and added to the `BASE` features. This attribute list contains the `CANDIDATES` information that was tested against the `BASE` feature. The `CANDIDATES` information corresponds with the `BASE` features, which

include the attribute `distance`, `angle`, `closest_base_x`, `closest_base_y`, `closest_candidate_x`, and `closest_candidate_y`.

Note Not all CANDIDATES' attributes will appear in the TESTED_CANDIDATES_LIST. There is a preliminary test whereby the distance between the CANDIDATE's bounding box and the BASE's bounding box is considered. Only features for which this distance is less than or equal to `<maxDistance>` will have their real distance from the base calculated, and these features' attributes will appear in the TESTED_CANDIDATES_LIST.

CLOSE_CANDIDATES_LIST has similar behavior to TESTED_CANDIDATES_LIST. The list that this clause creates will contain the attributes from all candidates within `<maxDistance>` of the BASE in question.

The CANDIDATES_LIST clauses also change the way that CANDIDATE feature attributes are copied to the BASE feature. Normally the BASE feature attributes are replaced by all of the CANDIDATE feature attributes. However, if both the BASE and CANDIDATE have an attribute with the same name, then the value of the CANDIDATE attribute replaces the existing BASE attribute value. If either CANDIDATES_LIST clause is specified, then the BASE attributes take precedence, so if both the BASE and CANDIDATE have an attribute with the same name, then the value of the BASE attribute will be preserved. Only new CANDIDATE attributes will be added to the BASE, but all of the CANDIDATE attributes will appear in the requested CANDIDATES_LIST(s).

Hint: To get the distance from a given BASE to all CANDIDATE features, use a very large number for `<max distance>` and specify CLOSE_CANDIDATES_LIST.

The GROUP_BY clause specifies the attributes used to group the base and candidate features. Each group of features is processed through the ProximityFactory independently of other groups. If no GROUP_BY clause is specified, then all features fall into the same group. Also note that the GROUP_BY clause cannot be used in conjunction with the SPATIAL_MATCH of ENVELOPE_INTERSECT.

If the FME_GEOMETRY_HANDLING directive is set to "yes" in the mapping file, arcs and ellipses will have their boundaries considered in distance calculations; otherwise, their centerpoints will be used.

Assumptions

None.

Output Tags

The `ProximityFactory` supports the following output tags.

Tag	Description
<code>MATCHED</code>	The features that result from a successful proximity match. These features have the geometry of the <code>BASE</code> feature with the attributes of the <code>CANDIDATE</code> feature added.
<code>UNMATCHED_BASE</code>	The <code>BASE</code> features for which there is no <code>CANDIDATE</code> feature within <code><maxDistance></code> .
<code>UNMATCHED_CANDIDATE</code>	The <code>CANDIDATE</code> features that are not the closest to any <code>BASE</code> feature.

Example

The example below matches each input point with its closest line, transferring the line attributes to each point. The lines and the points are, therefore, associated with common attribute values. The point features may be Global Positioning System (GPS) location readings and the line features may be the road network.

```

FACTORY_DEF * ProximityFactory                                \
  INPUT CANDIDATE FEATURE_TYPE * mapinfo_type mapinfo_polyline \
  INPUT BASE FEATURE_TYPE * mapinfo_type mapinfo_point         \
  SPATIAL_MATCH CLOSEST 10.0                                   \
  OUTPUT MATCHED FEATURE_TYPE * mapinfo_type mapinfo_point     \
  OUTPUT UNMATCHED_BASE FEATURE_TYPE * @Log (UNMATCH_BASE)     \
  OUTPUT UNMATCHED_CANDIDATE FEATURE_TYPE *                    \
    @Log (UNMATCHED_CANDIDATE)

```


PythonFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```

FACTORY_DEF <ReaderKeyword> PythonFactory
[FACTORY_NAME<factory name>]
[INPUT FEATURE_TYPE <feature type>
[<attribute name> <attribute value>]*
[<feature function>]* ]
SYMBOL_NAME<python symbol name>
[SOURCE_CODE<source code>]
[OUTPUT PYOUTPUT FEATURE_TYPE <feature type>
[<attribute name> <attribute value>]*
[<feature function>]* ]

```

Description

The PythonFactory is an FME Factory whose functionality is implemented using the Python language. A PythonFactory is implemented using a python object that is specified by some symbol name. The symbol name must be fully qualified, and reference a valid Python object.

Assumptions

Python 2.3 or 2.4 must be installed on the machine where the translation will run. The standard FME installation does not include a Python interpreter. In addition, users of the PythonFactory should be familiar with pyfme, the Python wrapper to the FME Objects API.

Clauses

Clause	Description	Optional
SYMBOL_NAME	The name of a Python Function, Class, or Object that will be used to implement the factory.	No
SOURCE_CODE	Python source code that is encoded as described in Substituting Strings in Mapping Files, in the <i>FME Fundamentals</i> manual. For internal FME use by the PythonCaller.	Yes

Output Tags

Tag	Description
PYOUTPUT	Features passed to FME via the pyoutput method.

Python Implementation Objects

Three types of objects can be used to implement the factory:

- Python Classes
- Python Functions
- Python Instance Objects

Python Class Implementation

Either new-style or old style classes can be used to implement the factory. When a class object is used to implement the factory, the PythonFactory will create an instance of the class at runtime. The PythonFactory protocol is used to receive features from FME.

PythonFactory Protocol Methods

Method	Remarks
<code>input(self, feature)</code>	Allows the implementation class to receive input features from FME. This method is only required if the factory needs to handle input features. The feature parameter is a pyfme FMEFeature object. See the pyfme documentation for more information. This parameter is not an in/out parameter, i.e. any changes to the feature will be lost unless the feature is output using the self.pyoutput method.
<code>close(self)</code>	Called when the factory receives the “allDone” signal from FME, which indicates that the factory will not receive any further features.

Bridge to FME

Each implementation class can send features “over the bridge” to FME using the pyoutput method. The pyoutput(self,feature) method will be dynamically added to the instantiated class at runtime.

Factory Styles

Python Class implementations support implementation of the following factory styles.

Method	Remarks
Single Input-Single Output	The class defines an input method which processes each feature and immediately sends it back to FME via the self.pyoutput method. The class does not need to implement a close method.
Single/Multiple Output	Generator classes receive no input from FME, and therefore do not need to implement the input method. The class implements a 'close' method that generates multiple new features that are output using the self.pyoutput method.
Single/Multiple Input Single/Multiple Output	A combination of the previous two styles, this class can take one or more input features, and output.

Python Function Implementation

Python Functions can be used to implement factories that perform very simple operations on one feature at a time. The functionality provided is very similar to @Python.

Functions implementing the factory must have a single parameter: an in/out feature parameter.

Note: In contrast to the input method on class implementations, the feature parameter on function parameters is in/out; the feature object parameter is always returned back to FME as factory output.

Python Object Implementation

Python Object implementation is a slightly more complex variation on Class implementations. This manner of implementation allows the same class to be reused with slightly different initialization of the class. For example, a factory class that interacts with a Web Service may require the class to be initialized with user credentials prior to use.

Example

Example 1: Factory with Python Function Implementation

Factory Definition

```

FACTORY_DEF * PythonFactory                                \
    INPUT  FEATURE_TYPE *                                  \
    SYMBOL_NAME    pythonExample1.exampleFunction          \
    OUTPUT PYOUTPUT *
```

Python Source Code

```

# Save this Python code in pythonExample1.py
#

import pyfme

logger = pyfme.FMELogfile()

def exampleFunction(feature):
    feature.setAttribute("example", "hello world")
    logger.log("Test log")
```

Example 2: Factory with Python Class Implementation

Factory Definition

```

FACTORY_DEF * PythonFactory                                \
    INPUT  FEATURE_TYPE *                                  \
    SYMBOL_NAME    pythonExample2.exampleClass             \
    OUTPUT PYOUTPUT *
```

Python Source Code

```

# Save this Python code in pythonExample2.py
#

class exampleClass(object):
    def __init__(self):
        #perform initialization here.
        self.count=0

    def input(self, feature):
        self.count+=1
        self.pyoutput(feature)

    def close(self):
        feature = pyfme.FMEFeature()
        feature.setAttribute("_count_", self.count)
        self.pyoutput(feature)
```

Example 3: Factory with Object Implementation

Factory Definition

```

FACTORY_DEF * PythonFactory
INPUT FEATURE_TYPE *
SYMBOL_NAME pythonExample3.exampleObject
OUTPUT PYOUTPUT *

```

Python Source Code

```

# Save this Python code in pythonExample3.py
#

exampleObject = exampleClass(42)

class exampleClass(object):
    def __init__(self,scaleFactor):
        #perform initialization here.
        self.count=0
        self.scaleFactor = scaleFactor

    def input(self,feature):
        self.count+= 1
        feature.scale(scaleFactor,scaleFactor,scaleFactor)
        self.pyoutput(feature)

    def close(self):
        feature = pyfme.FMEFeature()
        feature.setAttribute("_count_",self.count)
        self.pyoutput(feature)

```

QueryFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```
FACTORY_DEF <ReaderKeyword> QueryFactory
[FACTORY_NAME <factory name>]
[INPUT FEATURE_TYPE <feature type>
 [<attribute name> <attribute value>]*
 [<feature function>]*]*
[QUERY_INTERACTION (EQUALS|DISJOINT|TOUCHES|CROSSES|
WITHIN|OVERLAPS|CONTAINS|INTERSECTS|<any DE-9IM string>)]
[QUERY_WHERE_CLAUSE <where clause>]
[TARGET_TABLE_FIELD <attribute name>]
[QUERY_SQL_STATEMENT <SQL statement>]
[COMBINE_ATTRIBUTES (RESULT_ONLY|PREFER_QUERY|
PREFER_RESULT)]
[COMBINE_GEOMETRY (RESULT_ONLY|QUERY_ONLY|AGGREGATE)]
READER_TYPE <format name>
READER_DATASET <dataset specifier>
[READER_IDS <field name>]
[READER_PARAMS [<parameter> <value>]*]
[UNIQUE_KEYS <key name>]*]
[OUTPUT (QUERY|RESULT) FEATURE_TYPE <feature type>
 [<attribute name> <attribute value>]*
 [<feature function>]*]*
```

Overview

ORACLE VERSION	Any references to Oracle 8 <i>i</i> throughout this chapter are also applicable to Oracle 9 <i>i</i> .
-------------------	--

This factory performs a spatial query against any FME reader. It may also be used to perform non-spatial queries, in which case it acts as a convenient way to inject features from any source dataset into the middle of a factory pipeline.

For each feature this factory receives by way of its INPUT clause, it retrieves content from the specified dataset. The type query used to retrieve content from the dataset varies depends on how the QueryFactory is configured.

Content retrieved from the input dataset is output via the RESULT clause, while query features are outputted via the QUERY clause. Query features are outputted with an attribute called _matched_records, which indicates the number of result features which resulted from the query for that query feature.

The following rules are used to determine the type of query used:

- 1 If an SQL statement is specified via the `QUERY_SQL_STATEMENT` clause, then the SQL statement is used to query the input dataset. The SQL statement can only be used if the FME reader supports the custom SQL statements via the `fme_execute_sql` `setConstraints()` search type. If the `QUERY_SQL_STATEMENT` clause is specified, then the `QUERY_INTERACTION` and `QUERY_WHERE_CLAUSE` clauses are ignored.
- 2 If a spatial interaction is specified via the `QUERY_INTERACTION` clause, a spatial relationship query using the geometry of the input feature is used to query the input dataset. All features which satisfy the spatial relationship are returned. The set of spatial interactions supported by the `QUERY_INTERACTION` clause are the same as the `SpatialRelationshipFactory`.
In addition, if a `WHERE` clause is specified via the `QUERY_WHERE_CLAUSE` clause, this clause is used to further limit the dataset.
The `WHERE` clause can only be used in conjunction with a spatial relationship query if the FME reader supports the `fme_spatial_interaction` `setConstraints()` search type (see below).
- 3 If a `WHERE` clause is specified via the `QUERY_WHERE_CLAUSE` clause, this clause is used to filter the input dataset. A `WHERE` clause can only be used if the FME reader supports the `fme_all_features` `setConstraints()` search type.
- 4 If the input feature has any associated geometry, the geometry's bounding box will constrain the search, limiting the result of the query to be the set of features in the input dataset which are inside or touching the search envelope.
- 5 If the feature has no geometry, all features in the input dataset will be returned.

All parameters to the `QUERY_SQL_STATEMENT`, `QUERY_INTERACTION`, `QUERY_WHERE_CLAUSE`, `READER_TYPE`, `READER_DATASET`, `READER_IDS`, and `READER_PARAMS` clauses are treated as dynamic expressions rather than literal strings. That is, they are evaluated at the time a query feature is accepted into the factory. This allows the values actually used for these parameters to be the results of functions executed on each query feature that enters the factory.

For example, the following factory definition will read from the dataset specified by the `format`, `dataset`, and `ids` attributes from the input query feature:

```

FACTORY_DEF * QueryFactory                                \
  INPUT FEATURE_TYPE *                                    \
  READER_TYPE      &format                                \
  READER_DATASET   &dataset                                \
  READER_IDS       &ids *                                  \
  OUTPUT RESULT FEATURE_TYPE *
```

Unless changed on the `OUTPUT` clause, all features resulting from the query will have the feature type given them by the source dataset reader.

The `COMBINE_ATTRIBUTES` clause determines which attributes are returned on features emitted from the `OUTPUT` clause. Acceptable values for this clause are:
`RESULT_ONLY`: Only attributes originating from the input dataset are placed on the result features.

`PREFER_QUERY`: Attributes from the query feature are merged with the attributes originating from the input dataset. In the event of a conflict between attribute names, attribute values are taken from the query feature.

`PREFER_RESULT`: Attributes from the query feature are merged with the attributes originating from the input dataset. In the event of a conflict between attribute names, attribute values are taken from the input dataset.

The `COMBINE_GEOMETRY` clause determines which geometry is returned with features emitted from the `OUTPUT` clause. Acceptable values for this clause are:
`RESULT_ONLY`: Only geometry originating from the input dataset is placed on result features.

`QUERY_ONLY`: The geometry from the query feature is placed on all result features.

`AGGREGATE`: Each result feature contains an aggregate of the geometry originating from the input dataset and the geometry from the source feature.

Reader Specification

The `READER_TYPE` clause specifies the FME reader type from which to retrieve features. The `QueryFactory` may be used in conjunction with any of FME's readers.

If a spatial relationship query is being used against the input dataset, and the FME reader supports the `fme_spatial_interaction` `setConstraints()` search type, the spatial relationship query is optimized by executing the query directly against the input dataset. The spatial relationship query is otherwise transparently handled via an internal instance of a `SpatialRelationshipFactory`.

Otherwise, if a search envelope query is being used, readers that support the `SEARCH_ENVELOPE` directive (Oracle8i, ESRI Shape, and ESRI SDE, for example) will perform spatial envelope queries particularly efficiently, but other readers will also work.

The `READER_DATASET` keyword is used to specify the input dataset. It corresponds to the dataset normally specified for the given reader.

The optional `READER_IDS` clause may be used to select specific tables within a database dataset, or specific files within a directory-based dataset. If specified, the `TARGET_TABLE_FIELD` allows each query feature to specify which table or file should be searched by setting the attribute of the name to the desired table or file name.

Additional parameters may be given to the data readers using the `READER_PARAMS` clause. They are passed verbatim to the reader being used to perform the data retrieval.

The following example performs spatial interaction queries against an Oracle Spatial table. The input query feature supplies the geometry which will be used during spatial interaction testing. The result features contain geometry from the input dataset which passed the spatial interaction tests. Attributes are copied from the query feature to the result feature if they did not previously exist on the result feature.

```

FACTORY_DEF * QueryFactory \
  INPUT FEATURE_TYPE * \
  QUERY_INTERACTION INTERSECTS \
  COMBINE_ATTRIBUTES PREFER_RESULT \
  COMBINE_GEOMETRY RESULT_ONLY \
  READER_TYPE ORACLE8I \
  READER_DATASET kdb123a \
  READER_IDS ROADS \
  READER_PARAMS \
    USER_NAME fme \
    PASSWORD testsuite \
  OUTPUT RESULT FEATURE_TYPE * \

```

The following example performs spatial envelope queries against an Oracle8i Spatial table. The input query feature supplies the search envelope for the query.

```

FACTORY_DEF * QueryFactory \
  INPUT FEATURE_TYPE * \
  READER_TYPE ORACLE8I \
  READER_DATASET kdb123a \
  READER_PARAMS \
    SERVER_TYPE ORACLE8I \
    USER_NAME fme \
    PASSWORD fmepass \
  READER_DATASET kdb123a \
  READER_IDS ROADS \
  OUTPUT RESULT FEATURE_TYPE * \

```

Consecutive Queries

The `QueryFactory` performs a dataset read for each feature that enters its `INPUT` clause. If given more than one query feature, the factory may possibly return any feature in the dataset any number of times.

This effect can be avoided. The `UNIQUE_KEYS` clause specifies a set of attributes that uniquely identify a feature within the source dataset. If any unique keys are specified, the corresponding attribute values are inspected on each feature returned from the reader. Only the first feature with a given combination of attribute values will be emitted from the `QueryFactory`.

Example

The following mapping file performs two queries. The query features are created by the `CreationFactory` instances. The first tells the `QueryFactory` to access an Oracle database of roads to retrieve all features that interact with the rectangle (0.0,0.0) through (1.0,1.0). The second query feature specifies the same query on a MIF dataset.

Both datasets use the `ROAD_ID` attribute to uniquely identify the road. If any roads exist both within the Oracle database and the MIF files, they will have the same `ROAD_ID` value, and will thus only be emitted once for both queries.

Note that the Oracle login parameters (`SERVER_TYPE`, `USER_NAME` and `PASSWORD`) are sent to both readers, but are ignored by the MIF reader.

```
# -----
# Simple example of the QueryFactory.  It performs the same
# query on both an Oracle table and a MIF dataset.

READER_TYPE NULL
WRITER_TYPE NULL
NULL_DATASET null

# -----
# Create the ORACLE query feature

FACTORY_DEF * CreationFactory                                \
  2D_GEOMETRY 0 0 1 0 1 1 0 1 0 0                          \
  OUTPUT FEATURE_TYPE QueryFeature                          \
    Format      ORACLE8I                                     \
    Dataset     muni12                                       \
    IDs         ROADS                                        \
    Username    joe                                          \
    Password    blow                                         \
    ServerType  ORACLE8I                                     \

# -----
# Create the MIF query feature

FACTORY_DEF * CreationFactory                                \
  2D_GEOMETRY 0 0 1 0 1 1 0 1 0 0                          \
  OUTPUT FEATURE_TYPE QueryFeature                          \
    Format      MIF                                           \
    Dataset     "$(FME_MF_DIR)/my_municipality"              \
    IDs         ROADS                                         \

# -----
# Perform the queries. Note that both the MIF and Oracle readers
# are passed the Oracle login parameters (SERVER_TYPE, USER_NAME,
# and PASSWORD). The MIF reader will simply ignore the parameters
# it doesn't understand, so this won't cause a problem.

FACTORY_DEF * QueryFactory                                  \
  INPUT FEATURE_TYPE QueryFeature                            \
  READER_TYPE      &Format                                   \
```

```
READER_DATASET    &Dataset
READER_IDS        &IDs
READER_PARAMS     SERVER_TYPE &ServerType
                  USER_NAME   &Username
                  PASSWORD     &Password
UNIQUE_KEYS ROAD_ID
OUTPUT
```


RasterClippingFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> RasterClippingFactory          \
[FACTORY_NAME <factory name>]                               \
[INPUT CLIPPER FEATURE_TYPE <factory type>                  \
  [<attribute name> <attribute value>]*                       \
  [<feature function>]*]+                                     \
[INPUT CLIPPEE FEATURE_TYPE <factory type>                  \
  [<attribute name> <attribute value>]*                       \
  [<feature function>]*]+                                     \
[MULTICLIP (YES|NO|CLIPPERS_FIRST)]                         \
[MERGE_CLIPPER_ATTRIBUTES (YES|NO)]                         \
[CLIPPER_ATTR_PREFIX <attribute prefix value>]              \
[PRESERVE_CLIPPEE_EXTENTS (YES|NO)]                         \
[GROUP_BY [<attribute name>]+]                               \
[OUTPUT (CLIPPED_INSIDE|CLIPPED_OUTSIDE|                     \
  INSIDE|OUTSIDE|EXTRA_CLIPPER)                               \
  FEATURE_TYPE <feature type>                                 \
  [<attribute name> <attribute value>]*                       \
  [<feature function>]*]*

```

Overview

This factory is used to perform a geometric clipping operation on input raster features. It takes two types of features identified by the `CLIPPER` and `CLIPPEE` input tags.

Clipping features are identified by the tag `CLIPPER`. These features identify the area against which all `CLIPPEE` features are processed. Clippers can be polygon, donut, or aggregate features. Clippees must have raster geometry; they are unaffected by raster band and/or palette subselection.

Clipping will operate such that for each clippee, the following possible output will be generated:

- One `CLIPPED_INSIDE` feature for every clipper that intersects the clippee, and one `CLIPPED_OUTSIDE` feature
- One `INSIDE` feature
- One `OUTSIDE` feature

Clippees will be output untouched through the `OUTSIDE` port if they are disjoint from all clippers.

Clippees will be output untouched through the `INSIDE` port if they are contained in, covered by, or equal to any of the clippers.

Otherwise, each clipper that intersects the clippee will produce output through the `CLIPPED_INSIDE` port. If `PRESERVE_CLIPPEE_EXTENTS` is set to `no`, the extents of this raster will be equal to the bounding box of the portion of the

clipper within the clippee. Otherwise, if `PRESERVE_CLIPPEE_EXTENTS` is set to `yes`, the extents will be identical to those of the clippee. The remaining areas of the clippee that are disjoint from all clippers will be output through the `CLIPPED_OUTSIDE` port. The extents of this raster will be identical to those of the clippee.

The `MULTICLIP` option has the same effect as the `MUTICLIP` option of the ClippingFactory. Please consult the ClippingFactory documentation for more information on this keyword.

If `MERGE_CLIPPER_ATTRIBUTES` is specified, then whenever a clippee is found to be inside or clipped inside a clipper, the attributes of the clipper will be copied to the inside parts of the output features. The `CLIPPER_ATTR_PREFIX` directive specifies a string prefix to be added to the attributes of a `CLIPPER` that are copied to a clippee. If attributes of the same name already exist on the clippee, they will be overridden. If this prefix is not specified or is empty, the attributes will still be copied to the output features, but they will not overwrite any existing attributes of the same name.

If the `GROUP_BY` clause is given, then each clippee will only be clipped by clippers that contain the same values as itself in the attributes specified by this clause.

Assumptions

None.

Output Tags

The `RasterClippingFactory` supports the following output tags.

Tag	Description
<code>CLIPPED_INSIDE</code>	Areas of a clippee that intersect a clipper.
<code>CLIPPED_OUTSIDE</code>	Areas of a clippee that do not intersect any clipper.
<code>INSIDE</code>	Clippees that are contained in, covered by, or equal to a clipper.
<code>OUTSIDE</code>	Clippees that are disjoint from all clippers.
<code>EXTRA_CLIPPER</code>	Extraneous clippers that are received by the factory when <code>MULTICLIP</code> is not set to <code>YES</code> .

Examples

In the following example, an input raster will be clipped by features representing lots. Attributes from clippers will be merged on to clippee features, using the prefix `clipper_`. Additionally, `CLIPPED_INSIDE` features will have their extents reduced to the bounding box of the clipper within the clippee.

```

FACTORY_DEF * RasterClippingFactory \
FACTORY_NAME RasterClipper \
INPUT CLIPPER FEATURE_TYPE lots \
INPUT CLIPPEE FEATURE_TYPE city_raster \
MULTICLIP yes \
MERGE_CLIPPER_ATTRIBUTES yes \
CLIPPER_ATTR_PREFIX clipper_ \
PRESERVE_CLIPPEE_EXTENTS no \
OUTPUT INSIDE FEATURE_TYPE * \
OUTPUT CLIPPED_INSIDE FEATURE_TYPE * \
OUTPUT CLIPPED_OUTSIDE FEATURE_TYPE * \
OUTPUT OUTSIDE FEATURE_TYPE *
```


RasterCreationFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> RasterCreationFactory      \
[FACTORY_NAME <factory name>]                          \
[WIDTH <number of columns>]                            \
[HEIGHT <number of rows>]                              \
[GENERAL_CELL_TYPE <numeric or color>]                 \
[CELL_ELEMENT_TYPE <size of a cell>]                   \
[CREATE_PALETTE [(YES|NO)]]                            \
[BAND_COLOR_MODEL <color model>]                       \
[PALETTE_COLOR_MODEL <color model>]                   \
[PALETTE_KEY_TYPE <size of key>]                      \
[X_CELL_ORIGIN <floating-point number>]               \
[Y_CELL_ORIGIN <floating-point number>]               \
[X_SPACING <floating-point number>]                   \
[Y_SPACING <floating-point number>]                   \
[X_UPPER_LEFT_COORD <floating-point number>]         \
[Y_UPPER_LEFT_COORD <floating-point number>]         \
[NODATA_COLOR_VALUE <3comma-separated floating-point>] \
[MIN_COLOR_VALUE <3comma-separated floating-point>]   \
[MAX_COLOR_VALUE <3comma-separated floating-point>]   \
[NODATA_NUMERIC_VALUE <floating-point number>]       \
[MIN_NUMERIC_VALUE <floating-point number>]           \
[MAX_NUMERIC_VALUE <floating-point number>]           \
[NODATA_ALPHA_VALUE <floating-point number>]         \
[MIN_ALPHA_VALUE <floating-point number>]             \
[MAX_ALPHA_VALUE <floating-point number>]             \
[RASTER_TYPE <Single_Value> |                        \
    <Checkered_Pattern> | <Checkerboard>]              \
[OUTPUT CREATED FEATURE_TYPE <feature type>]          \

```

Overview

This factory creates a feature with a raster of the specified size with general values using the parameters supplied. It is useful for creating a very large image with a user-specified width and height. This factory can create one of three raster types:

- A *single-value raster* contains only one single numeric/color value throughout the whole raster.
- A *checkered-value raster* has alternating numeric/color values arranged in a gradient throughout the whole raster.
- A *checkerboard raster* has alternating blocks of minimum and maximum values arranged in a standard eight by eight checkerboard configuration. The number of rows and columns specify the total size of the raster, and the rows and columns per color block, which is variable. If the number of rows and columns specified are not multiples of eight, the leftover space will be filled

with either nodata values or the minimum value, if no nodata value was specified.

The *random-value* raster type is now deprecated and no longer supported.

WIDTH and HEIGHT are integer values starting from 1. WIDTH and HEIGHT virtually have no maximum values; however, the largest 4-byte integer size in a 32-bit machine limits the values of WIDTH and HEIGHT.

GENERAL_CELL_TYPE can be either Numeric or Color. If Numeric is chosen, NODATA_NUMERIC_VALUE, MIN_NUMERIC_VALUE, and MAX_NUMERIC_VALUE should be used. If Color is chosen, NODATA_COLOR_VALUE, NODATA_ALPHA_VALUE, MIN_COLOR_VALUE, MIN_ALPHA_VALUE, MAX_COLOR_VALUE, and MAX_ALPHA_VALUE should be used.

NODATA_NUMERIC_VALUE, or NODATA_COLOR_VALUE and NODATA_ALPHA_VALUE are optional.

CREATE_PALETTE is either 'Yes' or 'No'. If this parameter is not specified it will default to 'No'.

BAND_COLOR_MODEL is useful for setting the color type. Options are any one of:

- RGBA_4BYTE
- RGB_3BYTE
- RGBA_8BYTE
- RGB_6BYTE
- RED_1BYTE
- GREEN_1BYTE
- BLUE_1BYTE
- ALPHA_1BYTE
- GRAY_1BYTE
- RED_2BYTE
- GREEN_2BYTE
- BLUE_2BYTE
- ALPHA_2BYTE
- GRAY_2BYTE

CELL_ELEMENT_TYPE is useful for setting the color type. Options are any one of:

- Real_64_Bits
- Real_32_Bits
- UInt_64_Bits
- UInt_32_Bits
- UInt_16_Bits
- UInt_8_Bits
- Int_64_Bits
- Int_32_Bits
- Int_16_Bits

- `Int_8_Bits`

`PALETTE_KEY_TYPE` is useful for setting the palette's key type. Options are any one of:

- `UInt_8_Bits`
- `UInt_16_Bits`
- `UInt_32_Bits`

`PALETTE_COLOR_TYPE` is useful for setting the palette's value type. Options are any one of:

- `RGB_3BYTE`
- `RGBA_4BYTE`
- `RGB_6BYTE`
- `RGBA_8BYTE`
- `GRAY_1BYTE`
- `GRAY_2BYTE`
- `STRING`

`X_CELL_ORIGIN`, `Y_CELL_ORIGIN`, `X_SPACING`, `Y_SPACING`, `X_UPPER_LEFT_COORD`, and `Y_UPPER_LEFT_COORD` are geographic information. They are required to have some values. Particularly, `X_SPACING` and `Y_SPACING` cannot be lesser than or equal to zero.

`RASTER_TYPE` sets the kind of raster being generated.

- A `Single_Value` `RASTER_TYPE` will produce a raster of `MAX_COLOR_VALUE` with `MAX_ALPHA_VALUE` of `WIDTH` by `HEIGHT`. The values in the `_COLOR_VALUE` boxes must be in triple zero to one floating-point values separated by commas. The values in the `_NUMERIC_VALUE` boxes must be within the range of the `CELL_ELEMENT_TYPE`.
- A `Checkered_Pattern` `RASTER_TYPE` *d* by *c* will produce a raster with each pixel alternating between `MIN_COLOR_VALUE` and a gradient.
- A `Checkerboard` `RASTER_TYPE` will produce a raster with alternating blocks of minimum and maximum value pixels, with a total of eight blocks in the horizontal and vertical directions. If the number of rows and columns specified are not evenly divisible by eight, the raster will be padded with nodata values at the right-hand side and the bottom.

Note that when selecting greyscale color models, the minimum, maximum and nodata values used for the raster are the values of the red component for each parameter.

See below for a description of the other types of output clauses that are supported.

Assumptions

None.

Output Tags

Tag	Description
CREATED	The raster feature being created by the factory.

Example

In the example below, the raster in the feature is being set to have a single blue color throughout the whole raster, and the size of the raster is 200 by 100. This raster is of type RGBA with a cell depth of 4 bytes.

```

FACTORY_DEF * RasterCreationFactory           \
  FACTORY_NAME RASTERRGBCREATOR               \
  WIDTH 200                                   \
  HEIGHT 100                                  \
  GENERAL_CELL_TYPE Color                    \
  CREATE_PALETTE No                          \
  BAND_COLOR_MODEL RGB_4BYTE                 \
  X_CELL_ORIGIN 0.0                          \
  Y_CELL_ORIGIN 0.0                          \
  X_SPACING 1.0                              \
  Y_SPACING 1.0                              \
  X_UPPER_LEFT_COORD 0.0                     \
  Y_UPPER_LEFT_COORD 0.0                     \
  NODATA_COLOR_VALUE                         \
  MIN_COLOR_VALUE 0,0,0                      \
  MAX_COLOR_VALUE 0,0,1                      \
  NODATA_ALPHA_VALUE                         \
  MIN_ALPHA_VALUE 0                          \
  MAX_ALPHA_VALUE 1                          \
  RASTER_TYPE Single_Value                   \
  OUTPUT CREATED FEATURE_TYPE RASTERRGBCREATOR_CREATED

```

In the example below, the raster in the feature is being set to have a gradient numeric pattern throughout the whole raster, and the size of the raster is 100 by 100. This raster is of type Int32 with a cell depth of 4 bytes.

```

FACTORY_DEF * RasterCreationFactory           \
  FACTORY_NAME RASTERNUMERICCREATOR           \
  WIDTH 100                                   \
  HEIGHT 100                                  \
  CREATE_PALETTE No                          \
  GENERAL_CELL_TYPE Numeric                  \
  CELL_ELEMENT_TYPE Int32_Bits               \
  X_CELL_ORIGIN 0.0                          \
  Y_CELL_ORIGIN 0.0                          \
  X_SPACING 1.0                              \
  Y_SPACING 1.0                              \
  X_UPPER_LEFT_COORD 0.0                     \
  Y_UPPER_LEFT_COORD 0.0                     \
  NODATA_NUMERIC_VALUE                       \
  MIN_NUMERIC_VALUE 0,0,0

```



```
MAX_NUMERIC_VALUE 0,0,1
```

RASTER_TYPE Gradient

OUTPUT CREATED FEATURE_TYPE RASTERNUMERICCREATOR_CREATED

/

/

RasterEvaluationFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> RasterEvaluationFactory      \
  [FACTORY_NAME <factory name>]                          \
  [INPUT A|B FEATURE_TYPE <factory type>                \
    [<attribute name> <attribute value>]*                  \
    [<feature function>]*]*                                \
  [INTERPRETATION_LIST <list of band interpretations>]    \
  [EXPRESSION_LIST <list of band expressions>]           \
  [GROUP_BY [<attribute name>]+]                          \
  [OUTPUT RESULT FEATURE_TYPE <feature type>         \
    [<attribute name> <attribute value>]*                  \
    [<feature function>]*]*

```

Overview

This factory is used to evaluate expressions on each cell in a raster, such as algebraic operations or conditional statements.

The factory has two input ports: A and B. The cardinality of the input is required to be one of the following cases:

- One or more As, no Bs
- One A, one or more Bs

When both A and B features are provided, the single A input will be paired with each B input.

The **GROUP_BY** clause can be used to evaluate expressions on multiple pairs of rasters (i.e. multiple As and multiple Bs). Note that this clause is not applicable when only A input is provided; in that case, each raster is considered individually and there are no groupings.

The output of the factory will be a single raster feature per input pair. The output rasters will have n bands, where n is the number of interpretation/expression pairs, specified through the **INTERPRETATION_LIST** and **EXPRESSION_LIST** keywords.

The **INTERPRETATION_LIST** clause accepts a semicolon-delimited list of the interpretation for each output band. Valid interpretations are INT8, INT16, INT32, INT64, UINT8, UINT16, UINT32, UINT64, REAL32, REAL64, GRAY8, GRAY16, RED8, RED16, GREEN8, GREEN16, BLUE8, BLUE16, ALPHA8, ALPHA16. Additionally, the interpretation for a band may be left unspecified. In this case, the output interpretation will be whatever data type was used to perform the operation. If the clause is unspecified, all bands will have an automatically determined interpretation.

The `EXPRESSION_LIST` clause accepts a semicolon-delimited list of expressions describing how to calculate each output band. Please see the Operands section for the syntax of an expression.

As an example, if the input for `INTERPETATION_LIST` was “`REAL64; ; UINT32`” and the input for `EXPRESSION_LIST` was “`A[0] + B[0]; A[0] / 2; A[1] * 2`”, then the output would be a raster with a Real64 band, a band whose type was automatically determined, and a UInt32 band. The cell values in these bands would be calculated using the specified expressions.

When pairs of inputs are being operated on (i.e. the expression references both input A and B), the output feature will have all the attributes from both feature A and feature B. If the same attribute exists on both input features, then the attribute value from feature B will be preferred. Similarly, the output raster feature will have the properties of raster B.

When only a single input is being operated on (i.e. the expression only references input A), the feature attributes and raster properties will remain unchanged.

Note the following restrictions on input features:

- All input features must have raster geometry.
- All paired rasters must have the same number of rows and columns.
- Either all bands used in the same expression must have the same nodata value, or all bands used in the same expression must have no nodata value.
- No band may have a palette.

Operands

An expression consists of a combination of operands, operators, and parentheses. White space may be used between the operands, operators, and parentheses as it is ignored by the expression processor. Where possible, operands are interpreted as integer values; otherwise, operands will be treated as floating-point numbers. Floating-point numbers may be specified in any of the ways accepted by an ANSI-compliant C compiler, except that “I”, “F”, “l”, and “L” suffixes are not permitted in most installations. For example, all of the following are valid floating-point numbers: 2.1, 3., 6e4, 7.91e+16. Operands may be any of the following:

- A band from an input raster, specified using the syntax `A[i]`, where A indicates the input port (i.e. it may be either A or B) and i indicates the band index. Note that band indices are 0-based.
- A numeric constant, either integer or floating-point.
- The value of an attribute, using the syntax `A:attributeName`. The attribute's value is used as the operand. If the attribute name contains any characters that are not alphanumeric or underscores, the attribute name must be enclosed in quotation marks, e.g. `A:"attribute name"`. If the attribute

FME Functions and Factories

- •
•
•
•
•

474

•
•
•
•
•
•

- •
•
•
•
•

474

•
•
•
•
•
•

474

Operator	Description
	Bit-wise OR. Valid for integer operands only.
&&	Logical AND. Produces a 1 result if both operands are non-zero, 0 otherwise.
	Logical OR. Produces a 0 result if both operands are zero, 1 otherwise.

Functions

Several functions are available for usage in expressions.

One useful function is `if(c, x, y)`. This function returns:

- `nodata` if `c` is `nodata`
- `x` if `c` is non-zero
- `y` otherwise

Another useful function is `isnodata(x)`. This function returns 1 if `x` is `nodata`, 0 otherwise.

Finally, the following C math functions can be used:

<code>acos</code>	<code>cos</code>	<code>abs</code>	<code>sinh</code>	<code>asin</code>
<code>cosh</code>	<code>log</code>	<code>sqrt</code>	<code>atan</code>	<code>exp</code>
<code>log10</code>	<code>tan</code>	<code>atan2</code>	<code>floor</code>	<code>pow</code>
<code>tanh</code>	<code>ceil</code>	<code>fmod</code>	<code>sin</code>	

Each of these functions invokes the C math library function of the same name. Refer to the *C Manual* entries for the library functions for details on what they do and what arguments are valid.

Nodata Behaviour

Any operator or function applied to a `nodata` value will result in `nodata`. For example `(nodata + 1) = nodata`.

The exception to this is the `isnodata` function, described in the Functions section.

`Nodata` may also be produced due to any invalid operands. Examples of these include:

- Division by zero.

- Modulus by zero.
- `sqrt` of a negative number

Additionally, the keyword `NODATA` may be used in expressions. This keyword will be considered as the `nodata` value of the bands being evaluated. For example, it could be used in an expression like `if(A[0]==0, NODATA, B[0])`.

Assumptions

None.

Output Tags

The `RasterEvaluationFactory` supports the following output tag.

Tag	Description
RESULT	The raster created by evaluating the specified expressions.

Examples

In the following example, one raster at a time will be evaluated. For each raster input, the output will be a raster with a single `Real64` band.

```
FACTORY_DEF * RasterEvaluationFactory \
FACTORY_NAME RasterExpressionEvaluator \
INPUT FEATURE_TYPE A my_raster \
INTERPRETATION_LIST "REAL64" \
EXPRESSION_LIST "A[0] * sqrt(2)" \
OUTPUT RESULT FEATURE_TYPE RASTEREXPRESSIONEVALUATOR_OUTPUT
```

In the following example, pairs of rasters will be considered simultaneously. The output raster will have `Red8`, `Green8`, and `Blue8` bands which are computed as the average of the corresponding bands from the input rasters.

```
FACTORY_DEF * RasterEvaluationFactory \
FACTORY_NAME RasterExpressionEvaluator \
INPUT FEATURE_TYPE A my_raster_1 \
INPUT FEATURE_TYPE B my_raster_2 \
INTERPRETATION_LIST "RED8; GREEN8; BLUE8" \
EXPRESSION_LIST "(A[0]+B[0])/2; (A[1]+B[1])/2; (A[2]+B[2])/2;" \
OUTPUT RESULT FEATURE_TYPE RASTEREXPRESSIONEVALUATOR_OUTPUT
```


RasterMergerFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> RasterMergerFactory           \
  [FACTORY_NAME <factory name>]                             \
  [INPUT FEATURE_TYPE <factory type>                       \
    [<attribute name> <attribute value>]*                    \
    [<feature function>]*]**                                 \
  [COUNT_ATTRIBUTE <attribute name>]*                       \
  [ACCUMULATE_ATTRIBUTES [yes|no]]                         \
  [GROUP_BY [<attribute name>+]*]                           \
  [OUTPUT MERGED FEATURE_TYPE <feature type>              \
    [<attribute name> <attribute value>]*                    \
    [<feature function>]*]**

```

Overview

This factory is used to merge a collection of overlapping input raster features into a single raster feature. The factory accepts a number of input raster features, each of which has one or more bands. The bands are removed from the input features and appended to a single output raster feature. The order of the input features and the order of the bands of the input features both determine the order of the bands in the output feature.

The input features may be partitioned into groups based on attribute values using the `GROUP_BY` clause and one raster feature is produced for each group. If the `GROUP_BY` clause is not specified, then all input features will be processed together and a single raster feature will be produced.

The properties of each raster within a group, such as the number of rows and columns, must match for the processing to proceed successfully. Each raster within a group must also overlap identically.

If the optional `COUNT_ATTRIBUTE` clause is given, a new attribute will be created on every output feature, containing the number of input features that were combined to form the output. If grouped, the output feature's count attribute will have the number of input features that merged within its group.

If `yes` is specified for the `ACCUMULATE_ATTRIBUTES` clause, the attributes of each input feature are merged onto the feature being output. If grouped, the attributes of each input feature within a group are merged onto the group's output feature. If `no` is specified for the clause, no user attributes will be carried across from the input features to the output features.

This factory accepts only features that have raster geometries and is unaffected by raster band and/or palette sub-selection.

Assumptions

None.

Output Tags

The `RasterMergerFactory` supports the following output tag.

Tag	Description
MERGED	The output feature created by merging the input features.

Example

In the following example, the input rasters of feature type `maps` are merged by `city_name` into the output raster feature. The attributes of the input rasters are accumulated and placed on the output, along with an additional attribute, `raster_count`, which holds the number of input raster features used to create the output feature.

```

FACTORY_DEF * RasterMergerFactory           \
FACTORY_NAME RASTERMERGER                   \
INPUT FEATURE_TYPE maps                     \
COUNT_ATTRIBUTE raster_count               \
ACCUMULATE_ATTRIBUTES yes                   \
GROUP_BY city_name                          \
OUTPUT MERGED FEATURE_TYPE RASTERMERGER_MERGED

```

RasterMosaicFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> RasterMosaicFactory          \
  [FACTORY_NAME <factory name>]                            \
  [INPUT_FEATURE_TYPE <factory type>                       \
    [<attribute name> <attribute value>]*                    \
    [<feature function>]*]**                                \
  [COUNT_ATTRIBUTE <attribute name>]                        \
  [ACCUMULATE_ATTRIBUTES [yes|no]]                         \
  [GROUP_BY [<attribute name>]+]                            \
  [NODATA_OVERWRITE [yes|no]]                              \
  [INTERPOLATION_TYPE_NAME [nearestneighbor|bilinear|      \
    bicubic|average4|average16]]                            \
  [COMPOSITE_USING_ALPHA_BAND [yes|no]]                   \
  [SNAPPING_TYPE [resample|offset]]                        \
  [OUTPUT MOSAICKED FEATURE_TYPE <feature type>         \
    [<attribute name> <attribute value>]*                    \
    [<feature function>]*]*

```

Overview

This factory is used to mosaic a collection of input raster features into a single raster feature. The factory accepts a number of input raster features, each of which has one or more bands. The selected bands are removed from the input features and mosaicked to form a new band in the single output raster feature. The order of the input features and the order of the selected bands of the input features both determine the order of the bands and the set of bands being mosaicked in the output feature. For example, there are two rasters – R1 and R2:

- R1 has four bands – B11, B12, B13, and B14 – but only B12 and B14 are selected
- R2 has three bands – B21, B22, and B23 – but only B21 and B22 are selected

The output raster will therefore have two mosaicked bands: the first one consists of B12 and B21, and the second one consists of B14 and B22. Furthermore, if R1 is the first feature going into the factory and R2 is the second, and if they overlap each other, then R2 will be on top of R1. B12 and B21 are a set of bands and B14 and B22 are another set of bands.

The input features may be partitioned into groups based on attribute values using the **GROUP_BY** clause and one raster feature is produced for each group. If the **GROUP_BY** clause is not specified, then all input features will be processed together and a single raster feature will be produced.

If the optional **COUNT_ATTRIBUTE** clause is given, a new attribute will be created on every output feature, containing the number of input features that were combined to form the output. If grouped, the output feature's count attribute will have the number of input features mosaicked within its group.

If `yes` is specified for the `ACCUMULATE_ATTRIBUTES` clause, the attributes of each input feature are merged onto the feature being output. If grouped, the attributes of each input feature within a group are merged onto the group's output feature. If `no` is specified for the clause, no user attributes will be carried across from the input features to the output features.

The option `NODATA_OVERWRITE` determines whether a nodata value on top can overwrite a real data value underneath when some input features occupy the same geographical space. If this option is set to `no`, the nodata value on top will be skipped and the real data value underneath will be preserved. If this option is set to `yes`, the nodata value on top will overwrite any real data value underneath. Choosing `yes` improves the mosaicking performance but may lose some real data. This option will not be taken into account if the `COMPOSITE_USING_ALPHA_BAND` option is set to `yes`.

The `COMPOSITE_USING_ALPHA_BAND` option specifies whether the alpha band on the input rasters should be used to blend values that are overlapping. If this option is set to `yes`, all input rasters must have exactly one alpha band, and the values from these alpha bands will be used to blend all the other bands' values in areas where overlap occurs, effectively achieving transparency. An alpha value of 0 means complete transparency, and an alpha value of 1 means complete opacity. If this option is set to `no`, mosaicking is performed without blending, with new overlapping values overwriting previous ones without concern for transparency. Choosing `yes` decreases the mosaicking performance but gives a more accurate result when rasters are partially transparent. When set to `yes`, this option overrides the `NODATA_OVERWRITE` option because nodata values are considered fully transparent. This option does not work with palettes.

The `SNAPPING_TYPE` option specifies how an unaligned raster will be snapped to the underlying reference grid. If the option is set to `RESAMPLE`, the raster will be resampled using the chosen `INTERPOLATION_TYPE_NAME`. If the option is set to `OFFSET`, the raster's origin will be offsetted to match the reference grid. Note that if the raster's X Cell Spacing or Y Cell Spacing is different from the reference grid, resampling will occur no matter what option is selected.

The option `INTERPOLATION_TYPE_NAME` gives five different interpolation choices ranging from high performance to high accuracy. The `nearestneighbor` interpolation method yields the highest performance with the least accuracy. The `bicubic` interpolation method yields the highest accuracy but may take longer to process the data. The `bilinear` interpolation method yields a medium result. The `average4` and `average16` interpolation methods have a performance similar to `bilinear` and can be useful for numeric rasters such as DEMs.

Although the RasterMosaicFactory has a few different options, it also has some limitations


```
FACTORY_NAME RASTERMOSAICKER
INPUT FEATURE_TYPE CADRG
COUNT_ATTRIBUTE raster_count
ACCUMULATE_ATTRIBUTES yes
GROUP_BY cadrg_toc_frame_boundary_rectangle_record_number
NODATA_OVERWRITE no
INTERPOLATION_TYPE_NAME bicubic
COMPOSITE_USING_ALPHA_BAND no
OUTPUT MOSAICKED FEATURE_TYPE RASTERMOSAICKER_MOSAICKED
```

RasterPyramidFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> RasterPyramidFactory      \
[FACTORY_NAME <factory name>]                          \
[INPUT FEATURE_TYPE <factory type>                  \
  [<attribute name> <attribute value>]*                \
  [<feature function>]*]*                               \
[SMALLEST_LEVEL_NUM_ROWS <number of rows>]            \
[SMALLEST_LEVEL_NUM_COLUMNS <number of columns>]      \
[NUMBER_OF_LEVELS <number of levels>]                 \
[USE_POWERS_OF_TWO [YES|NO]]                          \
[INTERPOLATION_TYPE_NAME [nearestneighbor|bilinear|   \
  bicubic|average4|average16]]                          \
[RASTER_INDEX_ATTRIBUTE <attribute name>]             \
[PYRAMID_LEVEL_ATTRIBUTE <attribute name>]            \
[NUM_PYRAMID_LEVELS_ATTRIBUTE <attribute name>]       \
[OUTPUT PYRAMIDS FEATURE_TYPE <feature type>      \
  [<attribute name> <attribute value>]*                \
  [<feature function>]*]*                               \

```

Overview

This factory creates a series of pyramid levels for each input raster feature. Either the smallest pyramid level size or the number of pyramid levels must be specified to determine the pyramid levels to be generated. Pyramid levels are created by resampling input rasters to various different resolutions.

If the **SMALLEST_LEVEL_NUM_ROWS** and **SMALLEST_LEVEL_NUM_COLUMNS** clauses are specified, then the smallest pyramid level generated will have the specified size. Each subsequent level will increase the number of rows and columns by a factor of two, until the size of the input raster is reached.

If the **NUMBER_OF_LEVELS** clause is specified, then the largest pyramid level generated will have half the number of rows and columns of the input raster. Each subsequent level will decrease the number of rows and columns by a factor of two, until the specified number of levels has been generated. Note that the input raster does not count towards the number of levels. For example, if **NUMBER_OF_LEVELS** is 3, then the output will consist of the input raster plus three generated levels.

If **USE_POWERS_OF_TWO** is set to YES, then the number of rows and columns in all generated levels will be powers of two. When the smallest pyramid level size is specified, the number of rows and columns in the smallest level will actually be the smallest powers of two that are greater than or equal to the specified values. When the number of levels is specified, the number of rows and columns in the largest pyramid level will be the greatest powers of two that are less than or equal to the number in the input raster. This is an optional clause; the default is NO.

Note that pyramid levels will not be generated if either the number of rows or columns is less than 2.

The `INTERPOLATION_TYPE_NAME` clause specifies the interpolation method to use when resampling to produce the pyramids. This clause is optional; it defaults to `NEARESTNEIGHBOR`.

If the `RASTER_INDEX_ATTRIBUTE` clause is specified, an attribute will be added to each output raster feature that identifies which raster it was created from. This index is zero-based, so all pyramid levels created from the first input raster will have a value of 0, all pyramid levels created from the second input raster will have a value of 1, etc. This clause is optional.

If the `PYRAMID_LEVEL_ATTRIBUTE` clause is specified, an attribute will be added to each output raster feature indicating its level in the pyramid. The input raster is considered to be the base of the pyramid, level 0. The second largest level will have a value of 1, the third largest level will have a value of 2, etc. This clause is optional.

If the `NUM_PYRAMID_LEVELS_ATTRIBUTE` clause is specified, an attribute will be added to each output raster indicating the number of levels, including level 0, in the pyramid to which it belongs. This clause is optional.

This factory accepts only features that have raster geometry and is unaffected by raster band and/or palette subselection.

Attributes will be carried across from the `INPUT` features to the respective `PYRAMIDS` features.

Assumptions

None.

Output Tags

The `RasterPyramidFactory` supports the following output tag.

Tag	Description
<code>PYRAMIDS</code>	Output raster features comprising the levels of the pyramid generated for each input feature.

Examples

In the following example, a pyramid will be created for input raster features such that the smallest pyramid level is 100x50. For example, if an input raster

feature is 640x480, then the generated levels will be 100x50, 200x100, and 400x200.

```

FACTORY_DEF * RasterPyramidFactory \
    FACTORY_NAME RasterPyramids \
    INPUT FEATURE_TYPE my_raster \
    SMALLEST_LEVEL_NUM_ROWS 50 \
    SMALLEST_LEVEL_NUM_COLUMNS 100 \
    USE_POWERS_OF_TWO NO \
    INTERPOLATION_TYPE_NAME nearestneighbor \
    OUTPUT PYRAMIDS FEATURE_TYPE RASTERPYRAMIDS_LEVELS

```

The following example is identical to the previous one, except that `USE_POWERS_OF_TWO` now has a value of `YES`. Now, the smallest pyramid level that will be generated is actually 128x64. For example, if an input raster feature is 640x480, then the generated levels will be 128x64, 256x128, and 512x256.

```

FACTORY_DEF * RasterPyramidFactory \
    FACTORY_NAME RasterPyramids \
    INPUT FEATURE_TYPE my_raster \
    SMALLEST_LEVEL_NUM_ROWS 50 \
    SMALLEST_LEVEL_NUM_COLUMNS 100 \
    USE_POWERS_OF_TWO YES \
    INTERPOLATION_TYPE_NAME nearestneighbor \
    OUTPUT PYRAMIDS FEATURE_TYPE RASTERPYRAMIDS_LEVELS

```

In the following example, three levels of pyramids will be created for input raster features. For example, if an input raster feature is 1280x768, then the generated levels will be 640x384, 320x192, and 160x96.

```

FACTORY_DEF * RasterPyramidFactory \
    FACTORY_NAME RasterPyramids \
    INPUT FEATURE_TYPE my_raster \
    NUMBER_OF_LEVELS 3 \
    USE_POWERS_OF_TWO NO \
    INTERPOLATION_TYPE_NAME bicubic \
    OUTPUT PYRAMIDS FEATURE_TYPE RASTERPYRAMIDS_LEVELS

```

The following example is identical to the previous one, except that `USE_POWERS_OF_TWO` now has a value of `YES`. Now, if an input raster feature is 1280x768, then the generated levels will be 1024x512, 512x256, and 256x128.

```

FACTORY_DEF * RasterPyramidFactory \
    FACTORY_NAME RasterPyramids \
    INPUT FEATURE_TYPE my_raster \
    NUMBER_OF_LEVELS 3 \
    USE_POWERS_OF_TWO YES \
    INTERPOLATION_TYPE_NAME bicubic \
    OUTPUT PYRAMIDS FEATURE_TYPE RASTERPYRAMIDS_LEVELS

```


RasterSplitterFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> RasterSplitterFactory          \
  [FACTORY_NAME <factory name>]                             \
  [INPUT FEATURE_TYPE <factory type>                         \
    [<attribute name> <attribute value>]*                     \
    [<feature function>]*]**                                  \
  [SPLIT_BY [BAND|BAND_AND_PALETTE|PALETTE]]                 \
  [RASTER_INDEX_FIELD _raster_index]                         \
  [BAND_INDEX_FIELD _band_index]                             \
  [PALETTE_INDEX_FIELD _palette_index]                       \
  [OUTPUT SPLIT FEATURE_TYPE <feature type>                 \
    [<attribute name> <attribute value>]*                     \
    [<feature function>]*]**

```

Overview

This factory separates each input raster feature into one or more output raster features based on the number of input bands and palettes. The option to separate the raster by bands only, by both bands and palettes or by palettes only, can be specified by the `SPLIT_BY` clause, explained below. The order of the input features and their bands and palettes will determine the order of the bands in the output feature.

The `SPLIT_BY` clause accepts one of three parameter values:

- BAND
- BAND_AND_PALETTE
- PALETTE

If the `SPLIT_BY` clause is specified as `BAND`, then each band on the input raster feature will be placed onto a unique output raster feature. Thus each output raster feature will have no more than one band, but each band may have multiple palettes.

If the `SPLIT_BY` clause is specified as `BAND_AND_PALETTE`, the bands are split in the same way as the `BAND` case, and additionally bands are constrained to having one palette. Thus, each output raster feature will have no more than one band and no more than one palette. This is the default functionality.

If the `SPLIT_BY` clause is specified as `PALETTE`, the palettes on each band are split into several bands on the same raster such that all bands are constrained to having one palette. Thus, each output raster feature will have no more than one palette per band, but may contain multiple bands per raster.

If the `RASTER_INDEX_FIELD`, `BAND_INDEX_FIELD` and/or `PALETTE_INDEX_FIELD` clause are specified, the attributes are added to the output raster to specify which raster/band/palette it originated from. To disable

this functionality, leave the parameter(s) empty. This mode only works with `SPLIT_BY BAND` and `BAND_AND_PALETTE` parameter values.

This factory accepts only features that have raster geometry and is unaffected by raster band and/or palette subselection.

Attributes will be carried across from the `INPUT` features to the respective `SPLIT` features.

Assumptions

None.

Output Tags

The `RasterSplitterFactory` supports the following output tag.

Tag	Description
<code>SPLIT</code>	Output features created from splitting the input feature(s).

Example

In the following example, the bands of the multi-band input raster feature of type `multi_band_raster` are split up and moved to the output raster features, which will each have one band and at most one palette.

```

FACTORY_DEF * RasterSplitterFactory \
  FACTORY_NAME RasterSplitter \
  INPUT FEATURE_TYPE multi_band_raster \
  SPLIT_BY BAND_AND_PALETTE \
  RASTER_INDEX_FIELD _raster_index \
  BAND_INDEX_FIELD _band_index \
  PALETTE_INDEX_FIELD _palette_index \
  OUTPUT SPLIT FEATURE_TYPE RASTERSPLITTER_SPLIT

```

RasterSubsetFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> RasterSubsetFactory           \
[FACTORY_NAME <factory name>]                             \
[INPUT FEATURE_TYPE <factory type>                       \
  [<attribute name> <attribute value>]*                     \
  [<feature function>]*]**                                  \
[NUM_TILE_ROWS <number of rows in each tile>]             \
[NUM_TILE_COLUMNS <number of columns in each tile>]       \
[NUM_HORIZONTAL_TILES <number of horizontal tiles>]       \
[NUM_VERTICAL_TILES <number of vertical tiles>]           \
[RASTER_INDEX_ATTRIBUTE <attribute name>]                 \
[TILE_ROW_ATTRIBUTE <attribute name>]                     \
[TILE_COLUMN_ATTRIBUTE <attribute name>]                 \
[OUTPUT TILES FEATURE_TYPE <feature type>             \
  [<attribute name> <attribute value>]*                     \
  [<feature function>]*]**

```

Overview

This factory splits each input raster feature into a series of tiles. The input raster can be split up by specifying either a tile size or a number of tiles.

The **NUM_TILE_ROWS** and **NUM_TILE_COLUMNS** clauses specify the tile size. These values must be greater than 0.

The **NUM_HORIZONTAL_TILES** and **NUM_VERTICAL_TILES** clauses specify the number of tiles into which the input rasters will be split. These values must be greater than 0.

If the **RASTER_INDEX_ATTRIBUTE** clause is specified, an attribute will be added to each output tile that identifies which raster it was created from. This index is zero-based, so all tiles created from the first input raster will have a value of 0, all tiles created from the second input raster will have a value of 1, etc.

If the **TILE_ROW_ATTRIBUTE** and **TILE_COLUMN_ATTRIBUTE** clauses are specified, attributes will be added to each output tile that identify the position of that tile in the input raster. These indices are zero-based. Tile row 0, tile column 0 corresponds to the upper-left tile.

This factory accepts only features that have raster geometry and is unaffected by raster band and/or palette subselection.

Attributes will be carried across from the **INPUT** features to the respective **TILES** features.

Assumptions

None.

Output Tags

The `RasterSubsetFactory` supports the following output tag.

Tag	Description
TILES	Output tiles created from splitting the input feature(s).

Examples

In the following example, input raster features will be split into a series of tiles that are 128 rows by 256 columns.

```

FACTORY_DEF * RasterSubsetFactory \
  FACTORY_NAME RasterTiler \
  INPUT FEATURE_TYPE my_raster \
  NUM_TILE_ROWS 128 \
  NUM_TILE_COLUMNS 256 \
  OUTPUT TILES FEATURE_TYPE RASTERTILER_TILES

```

In the following example, input raster features will be split into 50 tiles, with 10 tiles in the horizontal direction and 5 tiles in the vertical direction. For example, if an input raster feature is 5000 rows by 5000 columns, then each of the resulting tiles will be 1000 rows by 500 columns.

```

FACTORY_DEF * RasterSubsetFactory \
  FACTORY_NAME RasterTiler \
  INPUT FEATURE_TYPE my_raster \
  NUM_HORIZONTAL_TILES 10 \
  NUM_VERTICAL_TILES 5 \
  OUTPUT TILES FEATURE_TYPE RASTERTILER_TILES

```

RasterToVectorFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```

FACTORY_DEF <ReaderKeyword> RasterToVector          \
[FACTORY_NAME <factory name>]                        \
[PRESERVE_ATTRIBUTES (YES|NO)]                      \
[RASTER_ID_ATTRIBUTE <attribute name>]              \
[BAND_ID_ATTRIBUTE <attribute name>]                \
[EXTRACT_NODATA (YES|NO)]                          \
[CELL_GEOMETRY (POINTS|POLYGONS)]                  \
[INPUT FEATURE_TYPE <feature type>                  \
  [<attribute name> <attribute value>]*               \
  [<feature function>]* ]*                            \
[OUTPUT POINTS FEATURE_TYPE <feature type>          \
  [<attribute name> <attribute value>]*               \
  [<feature function>]* ]*

```

Overview

This factory takes numeric raster features and extracts their elements as three-dimensional vector features. One feature of `fme_point` or `fme_area` geometry will be output for each cell in the raster. Each point or polygon receives a raster ID and a band ID which correspond to the raster and band from which the vector feature originated.

Each cell or element in the raster will be output either as a 3D point feature or as a 3D polygon feature. Each feature represents a z value at a row and column intersection point in the input raster

The `PRESERVE_ATTRIBUTES` clause is used to specify that the output vector features should retain exposed user and format attributes from the input raster. Note that attributes for the raster ID and band ID will be overridden if they already exist. If this is the case, each output feature for a given raster and band will have similar values.

The `EXTRACT_NODATA` clause specifies whether vector features will be output for nodata cells in the input raster. If set to `NO`, no vector feature will be output for nodata cells in the raster. If set to `YES`, then a vector feature will be output for each nodata cell. The default is `YES`.

The `CELL_GEOMETRY` clause specifies whether to output feature of `fme_point` or `fme_area` geometry. If set to `POINTS`, one point feature will be created for each of the raster's cell, positionned at the cell's origin. If set to `POLYGONS`, then polygon features will be output, each one covering one of the raster's cell. The default is `POINTS`.

This factory accepts only features that have raster geometry. This factory accepts multibanded rasters. This factory supports sub selection of raster bands.

This factory does not accept rasters with selected bands that contain one or more palettes; palettes must be resolved first for selected bands.

Assumptions

None.

Output Tags

The `RasterToVectorFactory` supports the following output tags.

Tag	Description
POINTS	The resulting features of <code>fme_point</code> or <code>fme_area</code> geometry, one for each input raster cell.

Example

The following example takes an input feature type named `numeric_raster` and extracts the numeric values from each cell of its bands as polygons, preserving the exposed user and format attributes. The points will each receive a raster ID attribute and band ID attribute, named `_raster_num` and `_band_num`, respectively. Additionally, no points will be output for nodata cells. The original features are not output by the factory.

```

FACTORY_DEF * RasterToVectorFactory           \
  FACTORY_NAME RasterCellCoercer              \
  INPUT FEATURE_TYPE numeric_raster           \
  PRESERVE_ATTRIBUTES YES                     \
  RASTER_ID_ATTRIBUTE _raster_num              \
  BAND_ID_ATTRIBUTE _band_num                 \
  EXTRACT_NODATA NO                           \
  CELL_GEOMETRY POLYGONS                      \
  OUTPUT POINTS FEATURE_TYPE RASTERCELLCOERCER_POLYGONS

```


RecorderFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> RecorderFactory
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [FEATURE_FILE <filename>]+
  [MAX_FILE_BYTES <integer>]
  [COMPRESSION_LEVEL (0-9)]
  [PASSPHRASE <string>]
  [BYTE_ORDER (NATIVE|LITTLE_ENDIAN|BIG_ENDIAN)]
  [STORE_SCANNED_SCHEMA (YES|NO)]
MODE (RECORD|RECORD_PASS_THROUGH|PLAYBACK|PLAYBACK_AT_END)
  [OUTPUT RECORDED FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory allows FME to record all FME features that pass through the factory pipeline and store the results in a file on disk. The disk file, referred to as a *feature store*, can be used later to insert the same features back into the feature pipeline for this or any other mapping file. See the *FME Feature Store Reader/Writer* chapter in the *FME Readers and Writers* manual for more details on this format.

Applications of the `RecorderFactory` include:

- archiving raw, partially processed, or fully processed features in a format-neutral manner
- shipping feature data – especially from database-oriented systems – for processing at another site
- making a snapshot of data imported from a particular format to assist in fine-tuning mapping files

Note If you do not have a current version of FME and you are using the FFS format for data exchange or storage, you may receive this error message: No geometry mapping entry found for 'fme_raster' in metafile 'C:\Program Files\FME_1378\metafile\FFS.fmf'. Program Terminating. FFS files that are created with recent FME builds cannot be read by some earlier versions of Workbench (build 1378 and earlier).

The factory has the following modes of recording and playback:

- If the mode is `RECORD_PASS_THROUGH`, then the factory will record each feature it receives and immediately pass it through to the rest of the FME for further processing.
- If the mode is `RECORD`, then the factory will record each feature it receives but wait until the end of translation before it sends all recorded features

through to the rest of the FME for processing. This method is useful because if one of the features causes the FME translation to fail, the feature file will be complete and usable, whereas if a failure occurs when using the `RECORD_PASS_THROUGH` mode, the feature file may not be usable.

- If the mode is `PLAYBACK`, the features will be restored to the FME factory chain *before* any other features created in previous factories or read from an input data source.
- If the mode is `PLAYBACK_AT_END`, the features will be restored to the FME factory chain *after* any other features created in previous factories or read from an input data source.

Tip

The `FEATURE_FILE` clause specifies the name of the output file (in `RECORD` or `RECORD_PASS_THROUGH` modes) or input file (in `PLAYBACK` or `PLAYBACK_AT_END` modes). The filename parameter is an FME expression, which can include `&attribute` or `@Function()` components, when in `RECORD` or `RECORD_PASS_THROUGH` modes.

When the factory is in playback mode, multiple feature files can be specified. In this case, it plays back each of the files in sequence. If any of the playback files is a directory, then all files in that directory which have an extension of `.ffs` will be played back.

Note Due to run-time library limitations, there is a maximum size limit ($2^{32}-1$ bytes) for an FME recording file. If this limit is exceeded when creating a recording, the output file will be split and a second recording will be created. The second recording will have a sequence number appended to it. When the recording is played back, the second recording file is also automatically played back. For example, if `roads.ffs` was being created and it exceeded the limit, the extra features would be output to `roads_1.ffs`. Both recordings would automatically be played back at playback time.

The `MAX_FILE_BYTES` directive alters the size limit of individual recording files to the user-defined size.

Recording

This factory creates compressed feature stores. The user may specify a `COMPRESSION_LEVEL` value between 0 and 9. A lower compression level will result in faster operation for both reading and writing while a higher compression level will result in smaller file sizes. Compressed feature stores may also be created with a `PASSPHRASE` to encrypt the output file for additional security.

The default `COMPRESSION_LEVEL` is 6 and no `PASSPHRASE` is set.

The directive `BYTE_ORDER` indicates if the resulting file should be optimized for either `LITTLE_ENDIAN` or `BIG_ENDIAN` machines. (For example, the architecture of machines running Microsoft Windows is little endian, while the Solaris architecture is big endian.) The `BYTE_ORDER` of `NATIVE` means the file should be optimized for the type of machine it is currently running on. Note that all files created can be read back on machines of any byte order; the only issue is that reading back files optimized for the opposite byte order will take slightly longer. The default `BYTE_ORDER` is `NATIVE`.

The directive `STORE_SCANNED_SCHEMA` allows users to specify newly created FFS files to store scanned schemas automatically accumulated from all features passed into it. The default for this directive is `NO`.

Playing Back

This factory plays back both compressed and non-compressed feature stores. If a `PASSPHRASE` was used when creating the compressed feature store, it must be specified during playback.

Tip

If the feature store was recorded as `PORTABLE` using a version of the FME prior to *FME 2002*, the deprecated directive `PORTABLE` will be required; otherwise, this directive is no longer necessary.

Assumptions

None.

Output Tags

All features are emitted from the `RecorderFactory` with an output tag of `RECORDED`.

Example 1

In the following mapping file excerpt, a `RecorderFactory` is inserted at the head of the feature processing pipeline to store a copy of all features from the source data set into a feature store. The recorded data is intended to be passed to some clients, therefore the factory removes some confidential information before writing it to the feature store.

```
READER_TYPE SHAPE
WRITER_TYPE E00

SHAPE_DATASET "$(SourceDataset)"
E00_DATASET "$(DestDataset)"

# =====
# Record all of the SHAPE features read from the original file.
```

```

# In interests of privacy, we will strip out the "OWNER_NAME"
# attribute so that it won't be passed around with the feature
# store.

FACTORY_DEF SHAPE RecorderFactory                                \
    INPUT FEATURE_TYPE *                                       \
        @RemoveAttributes(OWNER_NAME)                         \
    FEATURE_FILE "$(STORE_FILE)"                               \
    MODE RECORD_PASS_THROUGH                                    \
    OUTPUT RECORDED FEATURE_TYPE *                              \

# =====
# Other feature processing...

FACTORY_DEF SHAPE ...
...
# =====

SHAPE_DEF building                                             \
    SHAPE_GEOMETRY      shape_polygon                          \
    BUILDIN_ID          number(5,0)                            \
    OWNER_NAME          char(50)                               \
    AREA                number(18,3)                           \

E00_DEF building_polygon                                       \
    BUILDIN_ID          number(5,0)                            \
    AREA                number(18,3)                           \

SHAPE building                                                 \
    BUILDIN_ID          %BUILDIN_ID                            \
    AREA                %AREA                                  \

E00 building_polygon                                           \
    e00_type            e00_poly                               \
    BUILDIN_ID          %BUILDIN_ID                            \
    AREA                %AREA                                  \

```

Example 2

The following mapping file processes the stored data as the sole input. It is exactly like the above file except it uses a `NULL` reader in place of the `SHAPE` reader, and flips the recording factory into `PLAYBACK` mode. The only lines that were changed from the previous mapping file were the reader type, reader keyword, and `RecorderFactory` mode. This technique allows even the most complex of mapping files to use recorded data with virtually no modification.

```

READER_TYPE NULL
READER_KEYWORD SHAPE
WRITER_TYPE E00

SHAPE_DATASET null
E00_DATASET "$(DestDataset)"

# =====
# Playback all of the SHAPE features previously recorded.

```

```
FACTORY_DEF SHAPE RecorderFactory \
    FEATURE_FILE "$(STORE_FILE)" \
    MODE PLAYBACK \
    OUTPUT RECORDED FEATURE_TYPE *

# =====
# Other feature processing...

FACTORY_DEF SHAPE ...

...
# =====

SHAPE_DEF building \
    SHAPE_GEOMETRY      shape_polygon \
    BUILDIN_ID          number(5,0) \
    OWNER_NAME          char(50) \
    AREA                number(18,3)

E00_DEF building_polygon \
    BUILDIN_ID          number(5,0) \
    AREA                number(18,3)

SHAPE building \
    BUILDIN_ID          %BUILDIN_ID \
    AREA                %AREA

E00 building_polygon \
    e00_type            e00_poly \
    BUILDIN_ID          %BUILDIN_ID \
    AREA                %AREA
```

ReferenceFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> ReferenceFactory
  [FACTORY_NAME <factory name>]
  [INPUT REFERENCER FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]+
  [INPUT REFERENCEE FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]+
  [GROUP_BY [<attribute name>]+]
  REFERENCEE_FIELDS [<field name>]+
  REFERENCER_FIELDS [<field name>]+
  REFERENCE_INFO (ATTRIBUTES|
    GEOM_BUILD_POLYS|
    GEOM_BUILD_AGGREGATES|
    GEOM_BUILD_LINES_FROM_POINTS|
    GEOM_AND_ATTR_BUILD_POLYS|
    GEOM_AND_ATTR_BUILD_AGGREGATES|
    GEOM_AND_ATTR_BUILD_LINES_FROM_POINTS)
  [DIMENSION (USEFIRST|2|3)]
  [LIST_NAME <field name>]
  [PROCESS_DUPLICATE_REFERENCES [(YES|NO)]]
  [MERGE_ATTRIBUTES [(YES|NO)]]
  [MANAGE_FME_TYPE [(YES|NO)]]
  [BUILD_INCOMPLETE_REFERENCES [(YES|NO)]]
  [DUPLICATE_ATTRIBUTES LIST]
  [DUPLICATE_ATTRIBUTES DELIMITED(<chars>)]
  [OUTPUT (COMPLETE|INCOMPLETE|REFERENCED|UNREFERENCED|
    NO_REFERENCES|NO_REFERENCEE_FIELD|
    DUPLICATE_REFERENCEE)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

Each ReferenceFactory accepts two kinds of features – *Referencer* and *Referencee* – and resolves references between them.

The ReferenceFactory merges the geometry of the REFERENCEE features into REFERENCER features: the REFERENCERS look for information from the REFERENCEEs.

For example, if you have a group of features that you want to combine with another group of features to get either their geometry or their attributes, or both, there is an attribute "key" value that each of them has that can be used to make the connection. REFERENCEEs are the features pointed to by a field of the REFERENCER features:

- When the `REFERENCE_INFO` directive is `GEOMETRY_*`, then the `REFERENCEES` are the features that contain the geometry. Note that the `REFERENCER` will lose its former geometry.
- When the `REFERENCE_INFO` directive is `ATTRIBUTES`, then the `REFERENCEES` contain attribution that will be joined to the attribution of the `REFERENCER` features. Note that any `fme_*` attributes present on the `REFERENCEE` features will not be transferred to the `REFERENCER` feature.
- When the `REFERENCE_INFO` directive is `GEOM_AND_ATTR_*`, then the `REFERENCEES` contain both geometry and attribution that will be joined to the `REFERENCER` features. Any geometry on the `REFERENCER` will be overwritten. Attributes on the `REFERENCER` may or may not be overwritten, depending on whether any `DUPLICATE_ATTRIBUTES` or `LIST_NAME` clause is given.

All features processed by the `ReferenceFactory` must share the same dimension. If `DIMENSION` is set to `USEFIRST`, then the dimension of the first feature going into the `ReferenceFactory` will be used to set the dimension of all features that follow. If `DIMENSION` is set to 2 or 3, then all features processed by the `ReferenceFactory` will have their features set to 2 or 3 respectively. The default setting is `USEFIRST`.

The input features may be partitioned by the `GROUP_BY` clause. If the `GROUP_BY` clause is specified, then references between features will only be resolved if they share a common value for the `GROUP_BY` attributes. If no `GROUP_BY` clause is specified, all features are processed together. A typical use is to `GROUP_BY multi_reader_id` when features are coming from a multi-reader to ensure that references are resolved within the correct set of features.

The factory performs the following operations:

- All `REFERENCEE` features for each `REFERENCER` feature are identified.
- The geometry from the identified `REFERENCEE` features is then combined to form the geometry of the `REFERENCER` feature. If `REFERENCE_INFO` is `*_BUILD_AGGREGATES`, the `ReferenceFactory` will create an aggregate of the geometries of the `REFERENCEE` features. (If there is only one `REFERENCEE` feature, then the `REFERENCER` geometry will be an aggregate with one element.) If `REFERENCE_INFO` is `*_BUILD_POLYS`, and the `REFERENCEES` consist exclusively of polygon and donut polygon features, then any common border segments will be removed. If `REFERENCE_INFO` is `*_BUILD_POLYS` and the `REFERENCEES` contain at least one non-donut or polygon feature, then the `ReferenceFactory` will form polygons and donuts from the `REFERENCEES`, and will join connected line segments of the `REFERENCEE` features before setting the geometry of the `REFERENCER` feature. In this last situation, the geometry may be an aggregate if several disjoint geometries were created. Lastly, if `REFERENCE_INFO` is `*_BUILD_LINES_FROM_POINTS`, the `ReferenceFactory` will connect the points of the `REFERENCEES` features into

lines. Any non-point features that are referenced will be ignored when building lines.

- The attributes from the identified `REFERENCEE` features are then combined to form the attributes of the `REFERENCER` feature. See the details for the `DUPLICATE_ATTRIBUTES` and `LIST_NAME` clauses below for more information.
- `REFERENCEE` features referenced are output by the output tag `REFERENCED`.
- `REFERENCEE` features not referenced by any `REFERENCER` feature are output by the output tag `UNREFERENCED`.
- `REFERENCER` features matched with *all* features they reference are output by the output tag `COMPLETE`.
- `REFERENCER` features that refer to one or more `REFERENCEE` features not found are output by the output tag `INCOMPLETE`. If `BUILD_INCOMPLETE_REFERENCERS` is specified, these features will have been merged with any that were actually found; otherwise, they will be untouched and any partial matches will be output as `UNREFERENCED`.
- `REFERENCEE` features that do not have any attributes identified by the `REFERENCEE_FIELDS` clause are output by the tag `NO_REFERENCEE_FIELD`.
- `REFERENCEE` features with the same value for the `REFERENCEE_FIELDS` as a previously encountered `REFERENCEE` are output via the `DUPLICATE_REFERENCEE` tag.
- `REFERENCER` features that do not have any value for the `REFERENCER_FIELDS` are output by the `NO_REFERENCES` tag.

The figure below illustrates the processing of the factory. The example illustrates the processing of three features: one `REFERENCER` and two `REFERENCEEs`. In this example, all `REFERENCEEs` are referenced and the `REFERENCER` references are all satisfied. After the `ReferenceFactory` has

Additional statistical information about the operation of the `ReferenceFactory` may be output by specifying `FME_DEBUG` `ReferenceFactory` in the mapping file. For details, search *Mapping File Debugging* in *FME Fundamentals help*.



Input Tags

Tag	Description
REFERENCEE	Features pointed to by one or more referencers.
REFERENCER	Features that point to one or more referencees.

The `ReferenceFactory` supports several configuration clauses.

REFERENCEE_FIELDS [*<field name>*]+ defines the fields that make up the REFERENCEE key.

If multiple fields are specified, then the key will be the concatenation of the fields in the order in which the name is specified. List attribute names may not be specified as part of the `REFERENCEE_FIELDS`.

`REFERENCER_FIELDS [<field name>]+` defines the fields within the `REFERENCER` object which, when combined, refer to a `REFERENCEE` object.

If multiple keys are specified, then the key values are concatenated in the order specified on the clause. The `ReferenceFactory` uses the resulting key to find the `REFERENCEE` object.

If a list attribute is specified for `REFERENCER_FIELDS`, then each of the list entries is used as a separate key to reference multiple `REFERENCEE` objects. If a list attribute is specified, no other fields can be specified. List attributes must contain a `{}` pair.

If either `LIST_NAME` or `PROCESS_DUPLICATE_REFERENCES` is given, then any number of `REFERENCEE` features will be allowed per `REFERENCER`. In the case of `LIST_NAME`, if attributes are to be combined, then all attributes of each `REFERENCEE` will be added to any `REFERENCER` that it matches, prefixed by “`<LIST_NAME>{index}.`” (index is zero for the first matching `REFERENCEE`, one for the second, and so on.) So if the third `REFERENCEE` for a given `REFERENCER` has an attribute named `myattr`, and `LIST_NAME` `mylist` was specified, then `mylist{2}.myattr` will be added to the `REFERENCER`.

If `PROCESS_DUPLICATE_REFERENCES` is set to `NO`, it means that any `REFERENCEE` features that have the same `REFERENCEE_FIELDS` are output using the `DUPLICATE_REFERENCEE` output. Only the first of the `REFERENCES` will be matched with a `REFERENCER`. If set to `YES`, then the duplicate `REFERENCES` are all matched with the corresponding `REFERENCER`.

If `MERGE_ATTRIBUTES` is specified, then existing attributes on a `REFERENCER` feature will never be overwritten by attributes of the same name on any `REFERENCEE` features.

When `MANAGE_FME_TYPE` is given, the factory will assign the appropriate value to the `fme_type` attribute on those features whose geometry is changed, according to the resultant geometry. Otherwise, `fme_type` remains unaltered.

The `BUILD_INCOMPLETE_REFERENCERS` clause specifies the factory’s behavior for `REFERENCERS` that fail to find a match with all their (multiple) `REFERENCEES`. If the clause is `YES`, any `REFERENCES` that do match are merged onto the `REFERENCER` and the `REFERENCEE` is output via the `REFERENCED` tag. The partially built `REFERENCER` is then output via the `INCOMPLETE` tag of the factory. If the clause is `NO`, any `REFERENCES` that are part of an incomplete referencer are output as `UNREFERENCED` and the `REFERENCER` is left untouched. The `REFERENCER` is again output as `INCOMPLETE`. When the clause is `NO` the only

way that a REFERENCEE is output as REFERENCED is if it is referenced by a REFERENCER that is complete.

The DUPLICATE_ATTRIBUTES clause directs the ReferenceFactory how to handle duplicate attributes when a referencer points to multiple referencees, due to the referencer key field being a list attribute. If not specified, then the attribute values of the last REFERENCEE are kept. If the value of the DUPLICATE_ATTRIBUTES clause is LIST, then the attributes that occur in multiple REFERENCEEs are stored in a list attribute of the form <attribute name>{0}, <attribute name>{1} etc. If the value of the DUPLICATE_ATTRIBUTES clause is DELIMITED(<chars>), then the attribute's values are stored in a single attribute named <attribute name> with the multiple entries separated by <chars>. <chars> is a string of 1 or more characters. Common values for <chars> are one of “ | , # ! ” and space.

Note You cannot have more than one REFERENCEE feature with the same key, assuming neither LIST_NAME nor PROCESS_DUPLICATE_REFERENCEES is specified. The DUPLICATE_ATTRIBUTES clauses are used if the REFERENCER has a list for the REFERENCER_FIELD. This way, there can be more than one REFERENCEE feature matched to one REFERENCER feature. To match more than one REFERENCER feature, you could use the ListFactory first, to group together REFERENCEE features.

If the FME_GEOMETRY_HANDLING directive is set to “yes” in the mapping file, the merged geometry preserves arcs, ellipses and text; otherwise, they appear as points in the merged geometry.

Output Tags

The ReferenceFactory supports the following output tags.

Tag	Description
COMPLETE	REFERENCER features for which all referenced features were present.
INCOMPLETE	REFERENCER features for which one or more referenced features were missing.
REFERENCED	REFERENCEE features referenced by one or more REFERENCER features.
UNREFERENCED	REFERENCEE features not referenced by any feature.
NO_REFERENCES	REFERENCER features that do not have any value for the REFERENCER_FIELD attribute.

Tag	Description
DUPLICATE_REFERENCEEE	REFERENCEEE features with the same value for the REFERENCEEE_FIELDS attributes. Note that no DUPLICATE_REFERENCEES will be output if LIST_NAME or PROCESS_DUPLICATE_REFERENCEES is specified.
NO_REFERENCEEE_FIELD	REFERENCEEE features with no value for the REFERENCEEE_FIELDS attributes.

Example 1

The following example creates a set of REFERENCER and REFERENCEE features. A ReferenceFactory then matches key attributes.

```

READER_TYPE NULL
NULL_DATASET null
WRITER_TYPE NULL

LOG_STANDARDOUT yes

# Create a single "master" feature that has geometry on it.
# This is our "referencer"

FACTORY_DEF * CreationFactory                                \
    OUTPUT FEATURE_TYPE master                              \
        id 1                                                 \
        @GeometryType(fme_point) @Log(OriginalMaster)      \
    2D_GEOMETRY 1000 1000

# Create a single "slave" feature. This one has the attributes
# the master needs.

FACTORY_DEF * CreationFactory                                \
    OUTPUT FEATURE_TYPE slave                                \
        myKey 1                                              \
        Attribute1 Value                                     \
        Attribute2 anotherValue

FACTORY_DEF * ReferenceFactory                               \
    INPUT REFERENCER FEATURE_TYPE master                    \
    INPUT REFERENCEE FEATURE_TYPE slave                      \
    REFERENCER_FIELDS id                                     \
    REFERENCEE_FIELDS myKey                                  \
    REFERENCE_INFO ATTRIBUTES                                \
    OUTPUT COMPLETE FEATURE_TYPE * @Log(CompleteMasters)

```

Tip

If you want to preserve a feature that has one particular attribute, use a TeeFactory to copy your feature, and then use @KeepAttributes to keep the key field(s). This is more efficient than using @RemoveAttributes() to remove unnecessary fields.

Example 2

The following example defines a `ReferenceFactory` that accepts three types of features: `LineWork`, `Lake`, and `Roads`. The factory then resolves the references and passes the `COMPLETE` and `REFERENCED` features out of the factory with their feature type unchanged.

Any `REFERENCER` features not able to resolve all references are output with a feature type of `INCOMPLETE`, and all `REFERENCEE` features not referenced are output with the feature type set to `EXTRA`. In this way, the factory performs all reference matching, as well as identifies referential problems in the input data.

```

FACTORY_DEF TABLE ReferenceFactory                                \
  INPUT REFERENCEE FEATURE_TYPE LineWork                        \
  INPUT REFERENCER FEATURE_TYPE Lake                            \
  INPUT REFERENCER FEATURE_TYPE Road                            \
  REFERENCEE_FIELDS lineID                                      \
  REFERENCER_FIELDS geometry{}                                  \
  REFERENCE_INFO GEOMETRY                                       \
  OUTPUT COMPLETE        FEATURE_TYPE *                         \
  OUTPUT INCOMPLETE      FEATURE_TYPE INCOMPLETE *             \
  OUTPUT REFERENCED      FEATURE_TYPE *                         \
  OUTPUT UNREFERENCED    FEATURE_TYPE EXTRA                    \

```

Tip

Notice the wildcard feature type (*) has on the `OUTPUT COMPLETE` and `OUTPUT REFERENCED` clauses. When the * is used as the feature type on an output line, it will not change the existing feature type. This allows `Lake` and `Road` features to be output by the same `OUTPUT` clause.

ReportFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> ReportFactory
[FACTORY_NAME <factory name>]
[INPUT FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*
[COLUMN <attribute name> [<column title>] <width>]+
[COLUMN_SEPARATOR <separator>]
[REPORT_TITLE <title>]
[REPORT_FILENAME <report filename>]
[NO_COLUMN_HEADERS]
[APPEND]
[SUPPRESS_EMPTY_REPORT]
[WRITE_UTF8_REPORT]
[OUTPUT REPORTED
  FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*

```

Overview

This factory outputs one line for each feature that enters it into a columnar report. The columns in the report are populated by attributes in the feature. The report may be routed to an external file or it can be dumped into the FME log file.

Assumptions

None.

Clauses

The `ReportFactory` makes use of several additional clauses to fully specify its operation.

Clauses	Description	Optional
COLUMN <attribute name> [<<column title>] <width>	Defines an output column in the report. If no column title is present, the attribute name will be used as the column title. If the attribute's value could not fit in the width given, then it will be truncated to that width. At least one column definition must be present. There is no limit on the total number of column definitions. Example: COLUMN zValue Z 8	No

Clauses	Description	Optional
COLUMN_SEPARATOR <separator>	The character used as the separator between columns in the report. Default: : Example: COLUMN_SEPARATOR " "	Yes
REPORT_TITLE <title>	The title to be output as the first line of the report. Default: None Example: REPORT_TITLE "Pt . Errors"	Yes
NO_COLUMN_HEADERS	Specifies that the headers normally written to the top of the report to name the report and its columns are to be suppressed.	Yes
APPEND	Specifies that the report is to be appended to the specified file if it already exists rather than replacing it.	No
SUPPRESS_EMPTY_REPORT	Specifies that if not features are processed by the factory, then no report is generated. Otherwise an empty report will be generated.	No
WRITE_UTF8_REPORT	Specifies the encoding of the report file. If this flag is set, then the file is encoded in UTF-8, otherwise it is encoded in the current system encoding.	No
REPORT_FILENAME <filename>	The file name for the report. If no file name is given, then the report will be output to the FME log file. Default: None (log file) Example: REPORT_FILENAME c : \rp.txt	Yes

Output Tags

The ReportFactory supports the following output tag.

Tag	Description
REPORTED	All features that enter the factory are output through this clause.

Example

The following example produces a report of the first and last x coordinate and some other attributes from all `ErrorLine` features. Notice that as the features enter the factory, attributes holding the first and last x coordinate are added.

```
FACTORY_DEF MULTI_READER ReportFactory      \  
  INPUT FEATURE_TYPE ErrorLine              \  
    1stXCoor @Coordinate(x,0)               \  
    lastXCoor @Coordinate(x,END)            \  
    numCoords @NumCoords()                  \  
  COLUMN errorNumber "Err #"                5  \  
  COLUMN errorCode   "Code"                 4  \  
  COLUMN numCoords   "Points"               5  \  
  COLUMN 1stXcoor    "First X"              7  \  
  COLUMN lastXcoor   "Last X"               7  \  
  COLUMN_SEPARATOR " | "                   \  
  REPORT_FILENAME "$(ErrorDirectory)/errorRep.txt" \  
  REPORT_TITLE "TRIM Revision Check Error Report" \  
  OUTPUT REPORTED FEATURE_TYPE *
```

SamplingFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> SamplingFactory           \
  [FACTORY_NAME <factory name>]                         \
  [INPUT FEATURE_TYPE <feature type>                   \
    [<attribute name> <attribute value>]*                \
    [<feature function>]*]*                             \
  SAMPLE_RATE <sample rate>                             \
  [OUTPUT (SAMPLED/NOT_SAMPLED)                       \
    FEATURE_TYPE <feature type>                         \
    [<attribute name> <attribute value>]*                \
    [<feature function>]*]

```

Overview

This factory samples the input features at a specified rate and divides them into two groups: those which are matched by the sampling rate, and those which come between the sampled features. The first input feature and each <sample rate> feature thereafter is considered a “sampled” feature.

However, when the sample rate is negative, then the sampling factory behaves differently. For example, if the sample rate is -1, then only the first feature will pass as sampled. Similarly, if sample rate is -2, then only the first two features will pass as sampled, and so on.

The `SamplingFactory` is useful to test mapping files being developed for large data sources as it can greatly reduce the number of features processed by the system.

Tip

- A sample rate of 1 passes all features through the factory. This can be used to apply a feature function to every translated feature.
- A sample rate of 0 tells the `SamplingFactory` to treat all features as unsampled features.
- A sample rate of -1 tells the `SamplingFactory` to pass only the first feature as sampled feature.

Note Older versions of this factory did not support any output clauses, and simply wrote out the sampled features. To maintain backward compatability, an instance of the `SamplingFactory` which has no `OUTPUT` clause will behave as if a clause of “`OUTPUT SAMPLED FEATURE_TYPE *`” has been specified. Note that this is non-standard FME behavior.

Assumptions

None.

Output Tags

The `SamplingFactory` supports the following output tags.

Tag	Description
SAMPLED	Features that occur at the specified <code><sample rate></code> are output with this tag.
NOT_SAMPLED	Features that come <i>between</i> the sampled features are output with this tag.

Example

The following example passes one out of every five features through for further processing by the FME. The other four features are deleted.

```
FACTORY_DEF Shape SamplingFactory \
SAMPLE_RATE 5
```

The second example uses the `@Count` function and a sampling rate of 1 to assign a unique number to each feature as it passes through the FME.

```
FACTORY_DEF SAIF SamplingFactory \
INPUT FEATURE_TYPE * featNum @Count(features) \
SAMPLE_RATE 1
```

SDE30QueryFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```

FACTORY_DEF <ReaderKeyword> SDE30QueryFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [SDE_SERVER_FIELD <field name>]
  [SDE_INSTANCE_FIELD <field name>]
  [SDE_DATASET_FIELD <field name>]
  [SDE_USERID_FIELD <field name>]
  [SDE_PASSWORD_FIELD <field name>]
  [SDE_VERSION_NAME_FIELD <field name>]
  [SDE_TARGET_TABLE_FIELD <field name>]
  [SDE_SEARCH_METHOD_FIELD <field name>]
  [SPATIAL_FILTER_TRUTH_FIELD <field name>]
  [WHERE_CLAUSE_FIELD <field name>]
  [CORRIDOR_DISTANCE_FIELD <field name>]
  [MAX_NUM_FIELD <field name>]
  [QUERY_MODE (QUERY|DELETE|UPDATE)]
  [OUTPUT_DUPLICATES (yes|no)]
  [SEARCH_ORDER (OPTIMIZE|SPATIAL_FIRST|ATTRIBUTE_FIRST)]
  [GET_SPATIAL_RELATIONS (yes|no)]
  [COMBINE_ATTRIBUTES (RESULT_ONLY|QUERY_ONLY|
    PREFER_RESULT)]
  [COMBINE_GEOMETRY (RESULT_ONLY|PREFER_RESULT|PREFER_QUERY
    |AGGREGATE)]
  [SILENT_MODE <value>]
  [REJECTED_PIPELINE_DIRECTORY <value>]
  [REMOVE_TABLE_QUALIFIER (yes|no)]
  [CLEAR_MISSING_DATA (yes|no)]
  [OUTPUT (QUERY|RESULT|MATCHED_RECORDS) FEATURE_TYPE
    <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory can only be used in conjunction with ESRI's Spatial Database Engine Version (SDE) 3.x/8.x/9.x. It is capable of performing queries with spatial and/or attribute components. Queries are defined by the features received on the input clause, allowing the FME to perform sophisticated SDE queries and deletions.

The SDE30QueryFactory is driven by input features that contain the specifics of the query to be performed. For example, a simple user interface could create such query features. These query features could then be run through the FME

either immediately or as a batch job when SDE activity from connected users is low.

A query is defined by each input feature received and has the following properties:

Tip

When debugging an `SDE30QueryFactory` specification, use the `MAX_NUM_FIELD` value to avoid consuming large amounts of system resources with erroneous searches.

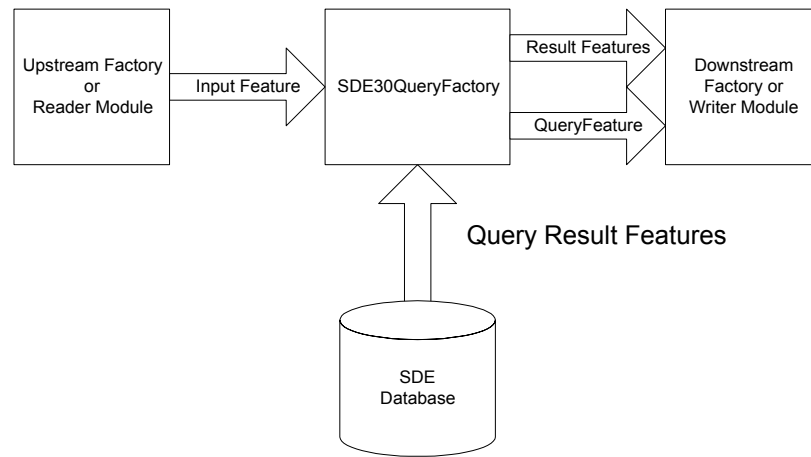
- Each feature defines the SDE upon which the query is to be performed enabling the FME to dynamically connect to any SDE on the network. Internally, the FME uses SDE connections sparingly to ensure there are no extraneous connections to any SDE being used by the FME.
- As with the SDE30 reader, a query may specify a multi-table join operation. See the *ESRI SDE 3.x/8.x Reader/Writer* chapter in the *FME Readers and Writers* manual for a description of this type of operation. The underlying mechanism is identical for both the SDE 30 reader and the `SDE30QueryFactory`.

In addition to simply retrieving features, the `SDE30QueryFactory` can also be used to delete features from the database as they are retrieved. This allows the FME to perform deletions from SDE 3.X based on attribute and spatial constraints. When operated in delete mode, the `SDE30QueryFactory` is still able to return the features that were deleted from the SDE for further processing.

Tip

Features that leave this factory are treated by FME as if they originated from the reader being used to drive the query.

The figure below illustrates the operation of the `SDE30QueryFactory`:



SDE30QueryFactory Processing

A feature flows into the `SDE30QueryFactory` either from the reader module or from an upstream factory. The features the `SDE30QueryFactory` accepts are called *query features* as they contain the parameters for the query to be performed. The `SDE30QueryFactory` uses these parameters to issue a query against the SDE.

As features are returned from the SDE, they are passed out of the `SDE30QueryFactory` without delay for immediate processing by the rest of the system.

After the last feature for a query has been returned by the SDE, the `SDE30QueryFactory` then passes the *query feature* to the rest of the FME system via the `OUTPUT QUERY` clause. The next query is started when the next query feature arrives.

Assumptions

ESRI SDE 3.x/8.x/9.x is installed on the database server you wish to retrieve data from.

Connecting to SDE

There are two possible ways to connect to SDE: connecting to the SDE service, which in turn communicates with the underlying database (a three-tier architecture), and connecting to the underlying database directly (a two-tier

architecture). With a direct connection (the two-tier architecture), SDE does not need to be installed and an ArcSdeServer license is only needed if writing to the database; reading does not require a license.

Regular Connection

The clauses needed for a regular connection are as follows:

Clause	Description	Optional
SDE_DATASET_FIELD	The attribute containing the name of the dataset upon which the query is performed. For SDE clients based on Oracle, any value will suffice.	No
SDE_SERVER_FIELD	The attribute containing the name of the server machine used to perform the query operation.	No
SDE_INSTANCE_FIELD	The attribute containing the name of the sde instance used to perform the query operation. Typically the value is port:5151 (or for older SDE's the typical value is esri_sde).	No
SDE_USERID_FIELD	The attribute containing the userid used to perform the query.	No
SDE_PASSWORD_FIELD	The attribute containing the user's password.	No
SDE_VERSION_NAME_FIELD	The attribute containing the version to which the FME is to connect. The version must already exist and the current user must have privileges set so that it can access the version. If the SDE_VERSION_NAME_FIELD directive is not used, then the factory connects to the version SDE.DEFAULT. This clause is only applicable when dealing with versioned tables and layers.	Yes

Direct Connection

The clauses needed to make a direct connection to SDE depend on the underlying database. In order to make a direct connection, the SDE must be of the same major version as the client libraries with which the SDE30QueryFactory was built. For example, a direct connection to an ArcSDE 9.1 instance could be made with an SDE30QueryFactory built using 9.0

libraries, but a connection could not be made to an ArcSDE 8.3 instance using the same SDE30QueryFactory.

Underlying Database	Clause	Description
Oracle (option 1)	SDE_INSTANCE_FIELD	The name of the attribute containing the value: sde:oracle or sde:oracle9i (for 9i connections to use the right driver)
	SDE_USERID_FIELD	<username>
	SDE_PASSWORD_FIELD	The name of the attribute containing the value: <password>@<Oracle Net Service Name>
Oracle (option 2)	SDE_INSTANCE_FIELD	The name of the attribute containing the value: sde:oracle://;local=<sqlnetalias>
	SDE_USERID_FIELD	The name of the attribute containing the value: <username>
	SDE_PASSWORD_FIELD	The name of the attribute containing the value: <password>
MS SQL Server	SDE_DATASET_FIELD	The name of the attribute containing the value: <database_name>
	SDE_INSTANCE_FIELD	The name of the attribute containing the value: sde:sqlserver:<SQL Server Instance Name> or sde:sqlserver:<SQL Server Instance Name>\<Named Instance> (for connecting to a named instance)
	SDE_USERID_FIELD	The name of the attribute containing the value: <username>
	SDE_PASSWORD_FIELD	The name of the attribute containing the value: <password>
DB2 (option 1)	SDE_DATASET_FIELD	The name of the attribute containing the value: <db alias name specified through DB2 Configuration Assistant>
	SDE_SERVER_FIELD	The name of the attribute containing the value: remote (if client application is remote, otherwise do not specify)

Underlying Database	Clause	Description
	SDE_INSTANCE_FIELD	The name of the attribute containing the value: sde:db2
	SDE_USERID_FIELD	The name of the attribute containing the value: <username>
	SDE_PASSWORD_FIELD	The name of the attribute containing the value: <password>
DB2 (option 2)	SDE_SERVER_FIELD	The name of the attribute containing the value: remote (if client application is remote, otherwise do not specify)
	SDE_INSTANCE_FIELD	The name of the attribute containing the value: sde:db2:<db alias name specified through DB2 Configuration Assistant>
	SDE_USERID_FIELD	The name of the attribute containing the value: <username>
	SDE_PASSWORD_FIELD	The name of the attribute containing the value: <password>
Informix	SDE_SERVER_FIELD	The name of the attribute containing the value: remote (if client application is remote, otherwise do not specify)
	SDE_INSTANCE_FIELD	The name of the attribute containing the value: sde:informix:<odbc data source name>
	SDE_USERID_FIELD	The name of the attribute containing the value: <username>
	SDE_PASSWORD_FIELD	The name of the attribute containing the value: <password>

The clause SDE_VERSION_NAME_FIELD can also be used to specify the attribute containing the version to use when making a direct connection. If not specified the version SDE.DEFAULT will be used.

Clauses

The following clauses are used to configure the `SDE30QueryFactory` for a retrieval operation:

Clauses	Description	Optional
<code>SDE_SERVER_FIELD</code>	The attribute containing the name of the server machine used to perform the query operation.	No
<code>SDE_INSTANCE_FIELD</code>	The attribute containing the name of the <code>sde</code> instance used to perform the query operation. Typically the value is <code>port:5151</code> (or for older SDEs the typical value is <code>esri_sde</code>).	No
<code>SDE_DATASET_FIELD</code>	The attribute containing the name of the dataset upon which the query is performed. This is a required field. For SDE clients based on Oracle, any value will suffice.	No
<code>SDE_USERID_FIELD</code>	The attribute containing the <code>userid</code> used to perform the query.	No
<code>SDE_PASSWORD_FIELD</code>	The attribute containing the user's password.	No
<code>SDE_VERSION_NAME_FIELD</code>	The attribute containing the version to which the FME is to connect. The version must already exist and the current user must have privileges set so that it can access the version. If the <code>SDE_VERSION_NAME_FIELD</code> directive is not used, then the factory connects to the version <code>SDE.DEFAULT</code> . This clause is only applicable when dealing with versioned tables and layers.	Yes
<code>SDE_TARGET_TABLE_FIELD</code>	If more than one table is to be specified in the field, then the table names must be separated by colons, ":". If there are joined tables, then the compound tables are described by including the secondary tables in parentheses. See the example outlined in <i>Multi-table Join Example</i> , in the <i>ESRI SDE 3.x/ArcSDE 8.x Reader/Writer</i> chapter of the <i>FME Readers and Writers</i> .	No
<code>SDE_SEARCH_METHOD_FIELD</code>	The attribute identifying the type of search method used to perform the query. If this is not specified then no spatial aspect of the query will be posed.	Yes
<code>SPATIAL_FILTER_TRUTH_FIELD</code>	This clause specifies the query feature attribute that indicates if features returned during the query will or will not satisfy the spatial constraint. This allows you to select features that are not contained by another, for example. The value contained in the query feature field must be either <code>TRUE</code> or <code>FALSE</code> . Default: <code>TRUE</code>	Yes

Clauses	Description	Optional
WHERE_CLAUSE_FIELD	The name of the feature's attribute that contains the WHERE clause used to perform attribute filtering. Note that double quotation marks are required around the WHERE clause, especially if you are querying across multiple fields: e.g. "OID = &OID and PARK = 'STANLEY'"	Yes
Corridor_distance_field	This clause is specified if a buffer is to be constructed around the search feature. The clause specifies the attribute containing the size of the buffer to be used. This buffered feature is then used as the search feature.	Yes
MAX_NUM_FIELD	This clause identifies the attribute which specifies the maximum number of features that can be returned by the query. This is used to ensure that an unexpectedly large number of result features are not accidentally returned.	Yes

Clauses	Description	Optional
QUERY_MODE	<p>Indicates if the features that satisfy the query are to be deleted from the SDE database, updated to the SDE database, or just retrieved from the SDE database.</p> <p>QUERY – features are retrieved from the database.</p> <p>DELETE – features are deleted as they are retrieved from the database. Currently this can only be used on tables that have a spatial column. To delete rows from tables that are not spatially enabled the user must use another method such as <code>@SDEsql()</code> or <code>@SQL()</code>.</p> <p>UPDATE – features are retrieved from the database. Then the attributes of the query feature are copied onto the result feature. If the same attribute exists on the query feature and the result feature, then the value on the result feature is overwritten with the value on the query feature. If the query feature contains some geometry, the geometry is copied over onto the result feature, overwriting any existing geometry originally found on the result feature. For the UPDATE mode to work, the tables being updated must either contain a spatial column or must be registered as multiversioned. If the clause <code>OUTPUT_DUPLICATES</code> is not specified, then if the same feature is retrieved from SDE database (by different query features), only the first update will be made. Subsequent updates will only be made if the <code>OUTPUT_DUPLICATES</code> clause is specified. WARNING! When copying features from one SDE to another SDE (i.e., when the query feature is a feature retrieved from a different SDE database), and when the result features come from a multiversioned table, the object ID field must be removed first from the query feature (if it exists), so that it doesn't overwrite the object ID field on the result feature.</p> <p>Range: <code>QUERY</code> <code>DELETE</code> <code>UPDATE</code></p> <p>Default: <code>QUERY</code></p>	Yes
OUTPUT_DUPLICATES	<p>If specified (with no argument or with an argument of <code>yes</code>), Yes the factory will not ensure that each spatial feature is updated only once and/or output only once (depending on whether the <code>QUERY_MODE</code> is set to <code>QUERY</code> or <code>UPDATE</code> and on whether the factory is outputting result features). The default behaviour for the factory is to keep track of the spatial features. In the event that a single factory performs multiple queries, it ensures that each spatial feature is only updated and/or output once.</p> <p>Specifying <code>OUTPUT_DUPLICATES</code> removes this check and results in duplicates being updated and/or output.</p>	Yes

Clauses	Description	Optional
	<p><code>sde30_identical</code>: the returned feature is identical to the query feature.</p> <p><code>sde30_area_intersects</code>: the returned feature intersects the query feature.</p> <p><code>sde30_interior_intersect</code>: the interior of the returned feature intersects the interior of the query feature.</p> <p><code>sde30_boundary_intersect</code>: the boundary of the returned feature intersects the boundary of the query feature.</p> <p><code>sde30_prim_lep_interior</code>: the end point of the query feature touches the interior of the returned feature.</p> <p><code>sde30_sec_lep_interior</code>: the end point of the returned feature touches the interior of the query feature.</p> <p><code>sde30_prim_contained</code>: the query feature is contained in the returned feature.</p> <p><code>sde30_sec_contained</code>: the returned feature is contained in the query feature.</p>	
COMBINE_ATTRIBUTES	<p>When the value specified is <code>RESULT_ONLY</code>, result feature attributes are based solely on the query results.</p> <p>When the value specified is <code>PREFER_RESULT</code>, result feature attributes are a combination of both the query results and query feature's attributes. If there is a conflict, attribute values taken from the query results.</p> <p>When the value specified is <code>PREFER_QUERY</code>, result feature attributes are a combination of both the query results and the query feature's attributes. If there is a conflict, attribute values are taken from the query feature.</p> <p>Default Value: NONE</p>	Yes

Clauses	Description	Optional
REMOVE_TABLE_QUALIFIER	<p>Specifies whether to keep or remove the table name prefix. Yes</p> <p>The default value is NO. Possible values are YES and NO. If the ArcSDE resides on a database (that is, MS SQL Server) where a specific value for database is set, then the full name for a table is</p> <p><database_name>.<owner_name>.<table_name>. If the ArcSDE is located on a database (that is, Oracle) that does not require the database field, then the full name of a table is <owner_name>.<table_name>. Setting this keyword to YES indicates that the reader should return the table name without any prefixes. This is useful when:</p> <ul style="list-style-type: none"> creating a workspace that will be passed on to another organization using the same table names, performing a translation to another database format but with a different user name, and when writing to a file-based format but not wanting the prefix in the name of the feature type. <p>When this keyword is set to YES during the generation of a mapping file or workspace, the source feature types will be the table names without any prefix; otherwise, they will contain the owner name as a prefix. It is recommended that this keyword not be changed in value after generating the mapping file/workspace as it is possible for no features to be successfully passed onto the writer (since the writer is expecting feature types with different names).</p> <p>Note that even when REMOVE_TABLE_QUALIFIER is set to YES, if the table is owned by a user other than the current user, the <owner_name> prefix will not be dropped so that the reader will find the correct table; however, the <database_name> prefix will still be dropped.</p>	
CLEAR_MISSING_DATA	<p>Valid values for this clause are YES and NO. The default value is NO. When set to YES, attributes not found on the query feature will be set to NULL on the feature(s) to be updated in ArcSDE. Likewise, if the query feature has no geometry, then the corresponding feature(s) to be updated will be given a nil geometry. When this clause is set to YES, make sure that, where appropriate, your columns allow NULL values.</p>	

This table lists the valid search methods.

Search Method	Description
SDE_ENVELOPE	The envelope of the returned feature overlaps or touches the envelope of the search feature.

Search Method	Description
SDE_COMMON_POINT	The search feature shares at least one common point.
SDE_LINE_CROSS	The search feature and the returned feature have intersecting line segments.
SDE_COMMON_LINE	The search feature and the returned feature share one or more common line segments.
SDE_CP_OR_LC	The search feature and the returned feature have line segments that intersect or have a common point.
SDE_AI_OR_ET	The search feature and the returned area feature edge touch (ET) or their areas intersect (AI).
SDE_AREA_INTERSECT	The search feature and the returned feature's area intersect.
SDE_AI_NO_ET	The returned feature and search feature have intersecting areas with no edge touching. One feature is therefore contained in the other.
SDE_CONTAINED_IN	The search feature is contained in the returned feature. For area features, this is clear. If search feature is a line, then a linear feature will be returned when the search feature path is included in returned feature. If search feature is a point, then the search feature will be one of the returned features vertices.
SDE_CONTAINS	The returned feature is contained by the search feature. If both features are linear features, then the returned feature must lie on the search feature's path. Point features that lie on a search feature vertex are also returned.
SDE_CONTAINED_IN_NO_ET	The returned feature must be an area feature that does not touch or share a vertex with the search feature. The returned feature contains the search feature.
SDE_CONTAINS_NO_ET	The returned feature is contained within the search feature. The returned features cannot touch the edge of, or share a vertex with, the search feature.
SDE_POINT_IN_POLY	The returned feature contains the first point of the search feature.
SDE_IDENTICAL	The returned feature has the same feature type and geometry. This is used to find duplicate data.

Output Tags

The SDE30QueryFactory supports the following output tags.

Tag	Description
QUERY	The query feature that entered the factory.
RESULT	Each of the features that satisfied the query.
MATCHED_RECORDS	Indicates how many records matched the query feature.

Using Versioning with the SDE30 Reader, Writer, and QueryFactory

Database states will be created by the FME only when updating/inserting/deleting from a versioned table or layer. Therefore, only the SDE30 writer or an SDE30QueryFactory in DELETE or UPDATE mode is able to create a database state.

The SDE reader and the SDE30QueryFactory in QUERY mode will never create a database state. All changes made during a single translation to a specific version (on the same SDE), even if they were made by different SDE30QueryFactories or different SDE writers, are placed into one (and the same) child state. Versioning must be used when updating/inserting/deleting from a versioned table or layer. If versioning is not used in these cases, then an “Insufficient permissions” error will be generated and the translation stopped.

Example

The following example shows the SDE30QueryFactory being used to retrieve features. Notice the use of macros for specifying the SDE Connection information. This factory is used in delete mode so features are deleted from the SDE as they are retrieved.

```

FACTORY_DEF * SDE30QueryFactory                                \
  INPUT FEATURE_TYPE *                                         \
    @SupplyAttributes(sdeServer,    $_SDE3Server))           \
    @SupplyAttributes(sdeInstance,  $_SDE3Instance))         \
    @SupplyAttributes(sdeUserName,  $_SDE3UserName))         \
    @SupplyAttributes(sdePassword,  $_SDE3Password))         \
    @SupplyAttributes(sdeDataset,   $_SDE3Dataset))          \
    @SupplyAttributes(sdeVersionName, "test-version")         \
    @SupplyAttributes(targetTables,  "PARCELS(STRATA)")       \
    @SupplyAttributes(whereClause,   "LOT_ID=&LOT_ID")        \
  QUERY_MODE DELETE                                           \
  SDE_SERVER_FIELD      sdeServer                             \
  SDE_DATASET_FIELD     sdeDataset

```

```
SDE_USERID_FIELD      sdeUserName      \  
SDE_INSTANCE_FIELD    sdeInstance      \  
SDE_PASSWORD_FIELD    sdePassword      \  
SDE_VERSION_NAME_FIELD sdeVersionName  \  
SDE_TARGET_TABLE_FIELD targetTables    \  
WHERE_CLAUSE_FIELD    whereClause      \  
SILENT_MODE           25                \  
OUTPUT QUERY FEATURE_TYPE * @Count(Q1query) \  
OUTPUT RESULT FEATURE_TYPE * @Count(Q1cleanedup)
```


SectorFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> SectorFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [SITE_FIELD <field name>]
  [AZIMUTH_FIELD <field name>]
  [SECTOR_FIELD <field name>]
  [RADIUS <value>]
  [MAX_DISTANCE <value>]
  [INPUT_CLUSTERED [(YES|NO)]]
  [GROUP_BY <field name>]
  [OUTPUT (SECTOR|AZIMUTH_LINE|SITE_POINT|EXTRA_POINT|
    DISTANT_POINT|INCOMPLETE_POINT|ILLEGAL_GEOM)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

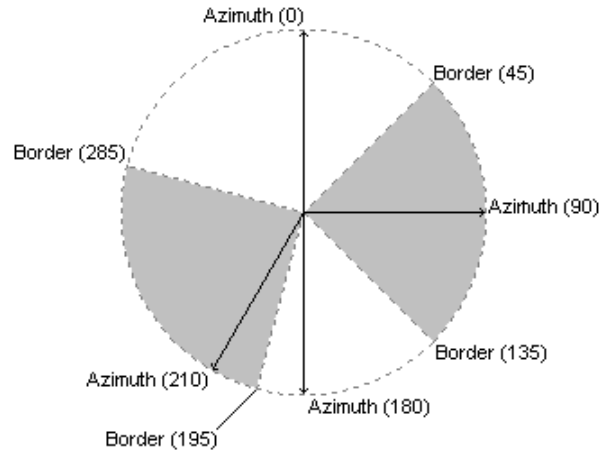
```

Overview

This factory takes as input a number of point features, and processes them based on specific attribute data. Each feature has the following:

- a site name (in the attribute named by `SITE_FIELD`),
- an azimuth (in the attribute named by `AZIMUTH_FIELD`): an angle measured in degrees clockwise from north (e.g., an azimuth of 90 points to east, 180 to south, etc.) which is the direction in which the feature is “pointing,”
- a radius (the value given in `RADIUS`, which may be a constant, or a per-feature value like `&Radius`), and
- optionally, a sector name (in the attribute named by `SECTOR_FIELD`). This is optional in the sense that `SECTOR_FIELD` is not a required clause; however, if it is specified, then all input features must contain this attribute.

Assuming the input features represent transmitters of some sort (hereinafter referred to as “sectors”), the `SectorFactory` generates polygons representing the approximate coverage area of each sector. At a given site, there may be one or more sectors. If there is only one, its area will be a circle with the given radius. If more than one sector exists at a site, each sector’s area will have a sector (“pie slice”) shape, again with the radius contained in the specified attribute. The border between adjacent sectors will be at the angle halfway between their azimuths. The following is an example of a site with four sectors:



The sector having the azimuth 0 produces the white region on top, the sector with azimuth 90 produces the grey region to the right, and so on.

The point coordinates for the sectors within a site must be within a given distance of each other. This distance is specified by the `MAX_DISTANCE` clause. Any input features a greater distance than this from their fellow sectors will be output on with the `DISTANT_POINT` tag. The point used for the center of the generated sectors will be the average of all the input coordinates.

If two or more sectors of the same site have the same azimuth, then the generated areas will behave as if there were only one instance of that azimuth, and multiple identically shaped sectors will be output.

If the `SECTOR_FIELD` clause is given, then attributes with the name supplied will contain the name of each sector. This can be used to ensure that the input does not contain duplicate sectors of the same name: any such input features will be output with the `EXTRA_POINT` tag.

Often input data will be in order:

```
Site;Sector;Azimuth;Radius
Sector1;A;0;5000
Sector1;B;30;5000
Sector1;C;220;5000
Sector2;A;0;5000
Sector3;A;270;5000
Sector4;C;0;5000
Sector4;A;120;5000
Sector4;B;240;5000
Sector5;A;0;5000
```

In a case like this, where all the sectors for a given site are guaranteed to be listed together in the input, the `INPUT_CLUSTERED` clause may be specified. This

enables the `SectorFactory` to save on memory usage by processing each site after all its features have entered the factory, rather than waiting for all features to be input.

The `GROUP_BY` clause can be given if the input contains multiple sets of sites and sectors, grouped according to any number of attributes. In this case, each set is dealt with independently. Since the factory must read in all features to separate them according to group, the `GROUP_BY` clause cannot be specified when `INPUT_CLUSTERED` is `YES`.

The arc portion of the output sectors is produced by the `@Arc` function, so the granularity of the line can be adjusted with the mapping file directive `FME_ARC_DEGREES_PER_EDGE`, which defaults to 5. (For instance, if a sector spans 35 degrees, by default its arc will contain seven edges.)

Any input feature that does not contain all the required attributes, or has null or invalid values (i.e., azimuth not in `[0, 360)` or nonpositive radius) will be output unchanged with the `INCOMPLETE_POINT` tag.

If the `FME_GEOMETRY_HANDLING` directive is set to `yes` in the mapping file, sectors are output as enhanced geometry polygons with arc segments or ellipses. Otherwise, sectors are output as polygons with only points and no arc segments.

Assumptions

None.

Clauses

The `SectorFactory` requires that a number of clauses be specified.

Clauses	Description	Optional
<code>SITE_FIELD <field name></code>	Specifies the name of the feature attribute that holds the site name. Default: None Example: <code>SITE_FIELD Site</code>	No
<code>AZIMUTH_FIELD <field name></code>	Specifies the name of the feature attribute that holds the azimuth. Default: None Example: <code>AZIMUTH_FIELD Azimuth</code>	No
<code>SECTOR_FIELD <field name></code>	Specifies the name of the feature attribute that holds the sector name. Default: None Example: <code>SECTOR_FIELD Sector</code>	Yes

Clauses	Description	Optional
RADIUS <value>	Specifies a value to use for the radius of each feature. This can be a constant, a function, or an attribute value (i.e., an attribute name prefixed with "&") Default: None Example 1: RADIUS @Value(Radius) Example 2: RADIUS 12.5	No
MAX_DISTANCE <value>	Specifies the maximum allowable distance between sectors of the same site. Any input features a greater distance than this from other features of the same site will be output under the tag EXTRA_POINT. Default: None Example: MAX_DISTANCE 0	No
INPUT_CLUSTERED [(YES NO)]	Specifies whether the input is clustered according to the attribute named with SITE_FIELD. If no argument is given or the argument is YES, then the factory will process the list of features read in so far every time it encounters one with a different site name (to save on memory usage). Cannot be YES when a GROUP_BY clause is present. Default: NO Example: INPUT_CLUSTERED YES	Yes
GROUP_BY [<field name>]+	Gives the names of one or more attributes by which to group the input. Each group is handled as though it were a separate factory. Default: All input belongs to the same group Example: GROUP_BY NetworkType	Yes

Output Tags

The SectorFactory supports the following output tags.

Tag	Description
SECTOR	Successfully constructed sectors are output with this tag.
AZIMUTH_LINE	Lines pointing in the direction of the azimuth and having the same length as the sector's radius are output with this tag.
SITE_POINT	A point for each site, located at the average of all the site's input points, is output with this tag. The attribute designated by SITE_FIELD will be copied onto SITE_POINT output features.

Tag	Description
EXTRA_POINT	This tag outputs points with a sector name that already belongs to a sector within the same site.
DISTANT_POINT	This tag outputs points that are a greater distance from other points in the same site than the tolerance specified in MAX_DISTANCE.
INCOMPLETE_POINT	This tag outputs features which do not contain all the attributes specified by the factory's _FIELD clauses, as well as those which have null values in any of these fields, contain a nonpositive radius, or have an azimuth less than 0 or greater than or equal to 360.
ILLEGAL_GEOM	Non-point input features are output with this tag.

Example

This example shows a SectorFactory which forces all sectors for a given site to have exactly the same point coordinate (MAX_DISTANCE 0) and which uses all of the output tags..

```
FACTORY_DEF * SectorFactory \
    INPUT FEATURE_TYPE data \
    SITE_FIELD Site \
    AZIMUTH_FIELD Azimuth \
    RADIUS &Radius \
    SECTOR_FIELD Sector \
    MAX_DISTANCE 0 \
    OUTPUT SECTOR FEATURE_TYPE sector \
    OUTPUT AZIMUTH_LINE FEATURE_TYPE azimuth \
    OUTPUT SITE_POINT FEATURE_TYPE site \
    OUTPUT EXTRA_POINT FEATURE_TYPE extra \
    OUTPUT DISTANT_POINT FEATURE_TYPE distant \
    OUTPUT INCOMPLETE_POINT FEATURE_TYPE incomplete \
    OUTPUT ILLEGAL_GEOM FEATURE_TYPE illegal
```

ServerJobSubmissionFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> ServerJobSubmissionFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  BLOCKING (yes|no)
  SERVER_NAME <field name>
  SERVER_PORT <integer>
  [USERNAME <field name>]
  [PASSWORD <field name>]
  REPOSITORY <field name>
  WORKSPACE_NAME <field name>
  [WORKSPACE_PARAMETERS <param>, <value> [<param>, <value>]*]
  SUBSECTION <field name>
  JOB_ID <field name>
  ATTR_PREFIX [<field name>]
  [OUTPUT (SUCCEEDED/FAILED)
    FEATURE_TYPE <feature type>
      [<attribute name> <attribute value>]*
      [<feature function>]*]*

```

Overview

This factory can be used to submit or run jobs on FME servers.

If **BLOCKING** is set to yes, then this factory waits until the job is processed by the server before proceeding. Upon completion, each response by the server will be added as an attribute to the feature, with the prefix specified in the **ATTR_PREFIX**. If **BLOCKING** is set to no, this factory can proceed as soon as the job is submitted to the server.

Workspace Parameters are used to specify the published parameters used by the workspace to be run or submitted to the server.

Each feature corresponds to one request made to the server. Features that successfully submit requests to the server will have a new attribute added (specified in **JOB_ID** clause) which denotes the job ID assigned by the server. Such features will be passed through the *SUCCEEDED* output. For any reason that the requests cannot be submitted to the server (invalid host, port number, workspace, or repository, etc.) the features will be output through the *FAILED* port.

Assumptions

None

Clauses

Clauses	Description	Optional
BLOCKING (yes no)	Controls if the factory waits for the job completion before proceeding. Default: None Example: BLOCKING yes	No
SERVER_NAME <field name>	The server that hosts the FME Server Default: None	No
SERVER_PORT <integer>	The port that the FME Server listens for incoming requests Default: None	No
USERNAME [<field name>]	The user name used to connect to FME server. Default: None	Yes
PASSWORD [<field name>]	The password used to connect to FME server. Default: None	Yes
REPOSITORY <field name>	The repository that contains the workspace. Default: None	No
WORKSPACE_NAME <field name>	The workspace to be submitted to FME Server. Default: None	No
WORKSPACE_PARAMETERS <param>, <value> [, <param>, <value>]*	The published parameter and value pairs used to run the workspace. Any ‘,’ character in the value must be replaced with <comma>. Default: None	Yes
SUBSECTION <field name>	The subsection name used by the server. Example: SUBSECTION SERVER_CONSOLE_CLIENT	No
JOB_ID <field name>	The attribute name to be added to the feature with the value being the job id assigned by the server.	No
ATTR_PREFIX [<field name>]	Each response from the server from running the workspace is added as new attribute to the feature, with the prefix specified here.	No

Output Tags

The ServerJobSubmissionFactory supports the following output tags

Tag	Description
SUCCEEDED	Features that successfully submitted the requests to the server.

Tag	Description
FAILED	Features that failed to submit the requests to the server.

ServerJobWaitingFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> ServerJobWaitingFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  SERVER_NAME <field name>
  SERVER_PORT <integer>
  [USERNAME <field name>]
  [PASSWORD <field name>]
  JOB_ID <integer>
  POLLING_INTERVAL <integer>
  ATTR_PREFIX [<field name>]
  [OUTPUT (SUCCEEDED/FAILED)
    FEATURE_TYPE <feature type>
      [<attribute name> <attribute value>]*
      [<feature function>]*]*

```

Overview

This factory can be used to wait until submitted FME Spatial ETL jobs are completely processed by an FME Server.

The **SERVER_NAME** and **SERVER_PORT** are the server host and port number that identify the FME Server where the jobs were submitted. The optional **USERNAME** and **PASSWORD** may be needed in order to gain access to the server.

The list of jobs to wait for is identified by the job IDs of the input features. When a job that it is waiting for is completed, it outputs the corresponding feature immediately.

The time interval for this transformer to wait between inquiries into the status of each job is specified by the Polling Interval. This is measured in seconds and can be entered as an integer value or an integer attribute. This parameter should not be set too small as this not only impacts the resources on the client, but also on the FME Server responding to each query. Generally use as large value as possible for this parameter. For example, if the job is expected to take about 20 minutes, then it is not efficient to set the Polling Interval to a few seconds.

Assumptions

None

Clauses

Clauses	Description	Optional
SERVER_NAME <field name>	The server that hosts the FME Server Default: None	No
SERVER_PORT <integer>	The port that the FME Server listens for incoming requests Default: None	No
USERNAME [<field name>]	The user name used to connect to FME server. Default: None	Yes
PASSWORD [<field name>]	The password used to connect to FME server. Default: None	Yes
JOB_ID <integer>	The ID that identifies one FME job. Default: None	No
POLLING_INTERVAL <integer>	Polling interval in seconds.	No
ATTR_PREFIX [<field name>]	Each response from the server from running the workspace is added as new attribute to the feature, with the prefix specified here.	No

Output Tags

The `ServerJobWaitingFactory` supports the following output tags

Tag	Description
SUCCEEDED	Features that successfully get requests to the server.
FAILED	Features that failed to get the requests to the server.
	An additional attribute <code>_job_failure_type</code> is added to the feature that holds one of the "Incomplete Parameters", "Connection or Server Problem" or "Translation Failed".

SmallworldGeometryFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> SmallworldGeometryFactory
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [ORIG_FEAT_TYPE_ATTR <attribute name>]
  [KEEP_SECTORS]
  [KEEP_MULTI_GEOM]
  [STROKE_ALL_ARCS]
  [FEATURE_TYPE_GENERATOR ( UNDERSCORE | ANGLE_BRACKETS ) ]
  [OUTPUT ( CHANGED | UNCHANGED ) FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory is used to flatten the geometry structure of features generated by the GE Smallworld reader. It accepts input Smallworld features which may contain an aggregate of geometry from zero or more geometric attributes and, unless the **KEEP_MULTI_GEOM** attribute is specified, it generates a separate feature for each contained geometry. Unless otherwise directed with the **KEEP_SECTORS** directive, this factory also joins together the sectors of each multiple-sector *sworld_chain* or *sworld_rope* geometric attribute to form a single line string, polygon or donut geometry for the attribute. Any input feature that contains no geometric attributes is passed through the factory unchanged.

An instance of this factory is all that is needed to process the features output from the Smallworld format reader so that they resemble the “normal” structure of geometric features passed through the FME.

Features whose geometry is changed by the factory – that is, its sectors were joined or its multiple geometry attributes were split into multiple features – are output with the *CHANGED* tag. If the factory had no effect on the input feature’s geometry, it will be output with the *UNCHANGED* tag.

Tip

The *UNCHANGED* tag indicates that the *geometry* of the feature was unchanged. It is possible for attributes to be defined or changed on output features tagged with this keyword.

The **ORIG_FEAT_TYPE_ATTR** keyword is used to instruct the factory to change the feature type of the output features to *<feature type>_<attr name>*, where *<feature type>* is the feature type of the original feature and *<attr name>* is the name of the geometric attribute contained in the output feature. In this case, the *<attribute name>* parameter to the **ORIG_FEAT_TYPE_ATTR** keyword names an attribute on the output feature into which the feature type of the

original input feature is written. Essentially, this feature makes a separate feature type for each geometric attribute defined on a Smallworld table instead of a feature type for the whole table.

If the keyword `FEATURE_TYPE_GENERATOR` is specified, then the feature types will be modified as this keyword directs. `UNDERSCORE` would modify the feature types as indicated above, `featureType_attrName` and `ANGLE_BRACKETS` would modify them like the following `featurType<attrName>`. The default is `UNDERSCORE`.

Assumptions

It is assumed that the input features are constructed exactly as the features generated by the Smallworld format reader. That is, it is assumed the geometry of the feature is:

- an aggregate of the feature’s geometric attributes’ geometries
- that chain and area geometries are themselves aggregates of the chain or area sectors
- that the `sworld_geometry{n}.*` attributes, as defined by the Smallworld reader, correspond to the actual structured geometry contained in the feature

If the input features are not constructed in this manner, the output from the `GeometryFactory` is undefined.

Output Tags

The `GeometryFactory` supports the following output tags.

Tag	Description
CHANGED	This tag is applied to output features whose geometry is somehow different from the feature which was input. This is normally the case if the feature contained more than one geometric attribute or if it contained any <code>sworld_chain</code> or <code>sworld_area</code> attributes.
UNCHANGED	This tag is applied if the feature’s geometry was not changed by the processing of the factory. This would be the case if there was no geometry on the input feature or if the <code>KEEP_MULTI_GEOM</code> tag was specified and the feature contained no <code>sworld_chain</code> or <code>sworld_area</code> attributes. Features tagged as <code>UNCHANGED</code> may still have attributes added to them to satisfy the geometry attribution described below and the ORIG_FEAT_TYPE directive, if specified.

Output Feature Attribution

In addition to the attribute defined by the optional `ORIG_FEAT_TYPE` directive, each output feature will likely have some geometry-related attributes defined on it. The only exceptions are:

- when the input feature does not contain any geometric attributes;
- when the `KEEP_MULTI_GEOM` directive is contained in the factory definition, in which case the output features have the same “aggregated multiple geometric attribute” form that the Smallworld format reader generates.

Except in the two instances just detailed, every attribute named `sworld_geometry{n}.<attrName>`, as assigned by the Smallworld reader, is copied into an attribute named “<attrName>” on the output feature. If the geometry of the feature is a single arc, circle, ellipse or spline, then each sector-related attribute will also be copied from

`sworld_geometry{n}.sector{m}.<attrName>`

to an attribute named “<attrName>”. Single-sector chain and area geometries are discussed in the following section.

In addition, every output feature containing a single geometry has an attribute named `sworld_geom_type` defined on it. This attribute indicates the type of geometry contained in the feature. It has one of the following values.

Value	Description
<code>sworld_point</code>	The feature has a point geometry, with all point-related attributes defined by the Smallworld format reader.
<code>sworld_text</code>	The feature has a point geometry, with all text-related attributes defined by the Smallworld format reader.
<code>sworld_area</code>	The feature has a polygon or donut geometry.
<code>sworld_line</code>	The feature has a simple line geometry, that is, a string of points.
<code>sworld_arc</code>	The feature has a point geometry, with all arc-related attributes defined by the Smallworld format reader.
<code>sworld_circle</code>	The feature has a point geometry, with all circle-related attributes defined by the Smallworld format reader.
<code>sworld_ellipse</code>	The feature has a point geometry, with all ellipse-related attributes defined by the Smallworld format reader.
<code>sworld_spline</code>	The feature has a simple line geometry, or string of points, with all spline-related attributes defined by the Smallworld format reader.

Deaggregation of Chains and Areas

Normally, the `GeometryFactory` combines the sectors of each `sworld_chain` or `sworld_area` geometry into a single line, polygon or donut geometry. Arcs, circles, and ellipses contained in the sector sequence are stroked into point sequences, in the same way an `@Arc()` function call would operate on such a geometry, before being inserted into the output line string. If a spline sector is encountered, its data points are strung together as if they were a simple line sector.

There are two cases in which the sectors of an `sworld_chain` or `sworld_area` geometry would not be combined into a single output geometry. The first is when the `KEEP_SECTORS` directive is specified in the factory definition. If this is specified, the original aggregate of sectors' geometry will remain on the output feature and the `sworld_sector{m}.<attrName>` attributes would not be copied to “<attrName>” attributes.

The second case, in which the sectors of an `sworld_chain` or `sworld_area` geometry would not be combined into a line string, polygon or donut feature, would be if the geometry contained a single sector and if that sector was an arc, ellipse, circle or spline. If this were to occur and the factory was operating in the “join sectors” mode – that is, `KEEP_SECTORS` was not specified – the output feature would have the deaggregated sector geometry in its original form and the `sworld_geom_type` attribute would contain a value of `sworld_arc`, `sworld_ellipse`, `sworld_circle` or `sworld_spline`, respectively.

Example

The following example creates features with different feature types depending on what geometric attributes are defined on them. For example, if a feature from the Smallworld table `building` is sent into the factory, with a text attribute `address` and a point attribute `location`, two features would be output from the factory:

- one with a feature type of `building_address` with the text geometry
- one with a feature type of `building_location` with the point geometry.

The original feature type `building` would be stored into the attribute `sworld_table_name`.

```
FACTORY_DEF SWORLD GeometryFactory\
    INPUT FEATURE_TYPE *           \
    ORIG_FEAT_TYPE_ATTR sworld_table_name \
    OUTPUT CHANGED FEATURE_TYPE *   \
    OUTPUT UNCHANGED FEATURE_TYPE *
```

SnappingFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```

FACTORY_DEF <ReaderKeyword> SnappingFactory
  [FACTORY_NAME <factory name>]
  [INPUT BASE|CANDIDATE FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  SNAP_TYPE (END_NODE|VERTEX)
  SNAP_TOLERANCE <distance>
  [EXTEND_LINES_TO_SNAP (ALWAYS|FORWARD_ONLY|NEVER)]
  [SAVE_SHORT_LINES [(YES|NO)]]
  [GROUP_BY [<attribute name>]+]
  [OUTPUT SNAPPED|UNTOUCHED FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory takes a series of features that match the input specification and performs snapping on the features that lie within the specified tolerance from other features that match the input specification. This factory can be used to perform some cleaning operations on data during a translation.

The factory operates in two modes:

- The input **BASE** clause is not specified, resulting in a snapping “free-for-all” between the **CANDIDATE** features where the features are within the **SNAP_TOLERANCE** value.
- The **BASE** clause is specified, and therefore no snapping is performed within the **BASE** features and within the **CANDIDATE** features. In this mode only **CANDIDATE** features are snapped to **BASE** features.

There are currently two types of snapping done by the factory, and this is specified by the **SNAP_TYPE** directive.

When the value for **SNAP_TYPE** is **END_NODE**, then features are processed as described below.

- The end nodes of linear features are considered as candidates for snapping. All other vertices will remain unchanged.
- Point features are also snapped if they are within the specified snapping tolerance.
- Polygon features are unchanged.

- The `EXTEND_LINES_TO_SNAP` clause may be specified, causing the factory to add another point to the line to complete the snap rather than move the existing vertex. If the clause is given the value `ALWAYS`, then a line is always added to complete the snap. If the value is `FORWARD_ONLY`, then the snapping is only done if the new point is closer to the existing endpoint than the previous point, thereby assuring that the line does not have a hard change in direction.

When `SNAP_TYPE` is `VERTEX`:

- Vertex-to-vertex snapping will be performed for vertices that are within `<tolerance>` of each other.
- Point features will be snapped together (or to a linear feature).
- Area features will be altered by this operation as its vertices are snapped.
- The `EXTEND_LINES_TO_SNAP` clause is ignored.

`SNAP_TOLERANCE` specifies the distance that the snapping occurs between features. This distance is in ground units.

If the `GROUP_BY` clause is given, then each feature will only be snapped to other features that contain the same values as itself in the attributes specified by this clause.

Any feature that undergoes dimensional collapse as a result of being snapped will be logged as "degenerate" and dropped. "Dimensional collapse" refers to a line or area that becomes a point, or an area that becomes a line.

If `SAVE_SHORT_LINES` is specified, any features entering the factory whose length is less than or equal to the tolerance will be treated specially: they will be output as `UNTOUCHED`, and other features (but not other short features) will be able to snap to them. If the option is not selected, features like this will collapse to a single point and will be dropped.

If the `FME_GEOMETRY_HANDLING` directive is set to "yes" in the mapping file, arcs are snapped as linear features, and ellipses are not snapped; otherwise, arcs and ellipses are both snapped as point features located at their respective center points.

Assumptions

None.

Output Tags

The `SnappingFactory` supports two output tags.

Tag	Description
SNAPPED	Features whose geometry is changed by the factory are output through this tag.
UNTOUCHED	Features that leave the factory without being changed are output through this tag.

Example

The following example illustrates the use of the `SnappingFactory` to snap together road segment endpoints.

```
FACTORY_DEF * SnappingFactory          \  
  INPUT CANDIDATE FEATURE_TYPE ROAD    \  
  SNAP_TYPE END_NODE                   \  
  SNAP_TOLERANCE 2.5                   \  
  EXTEND_LINES_TO_SNAP NEVER           \  
  OUTPUT SNAPPED FEATURE_TYPE *        \  
  OUTPUT UNTOUCHED FEATURE_TYPE *
```

SortFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> SortFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [SORT_BY [<attribute name> (NUMERIC|ALPHA)
    [(ASCENDING|DESCENDING)]+ ]
  [SORT_DIRECTION (ASCENDING|DESCENDING) ]
  [MEMORY_CACHE_SIZE <number>]
  [OUTPUT SORTED FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory sorts features according to the values of the specified attributes, as determined by the factory definition. For each attribute name, directions are given about how its value should be interpreted when it is compared. If *NUMERIC* is specified, then numeric comparisons will be done. If *ALPHA* is specified, then alphanumeric comparisons will be done. If no *SORT_BY* clause is specified, then the *SortFactory* will hold all features that enter the factory until all other features have passed. Then it will release the features into the factory pipeline. This is useful if, for example, you want to detain certain features until all input features have arrived, thereby ensuring that no additional processing is performed.

Once the factory is told to empty itself, it outputs all features it received in the order specified. Any *SORT_BY* clause that does not specify *ASCENDING* or *DESCENDING* will use the direction specified by *SORT_DIRECTION*. That is, the per-attribute sort direction overrides the global sort direction. By default, the *SORT_DIRECTION* is in ascending order.

This factory tries to balance speed and memory usage by storing some features in memory and some in a temporary disk file. The maximum number of features stored in memory can be set using the *MEMORY_CACHE_SIZE* clause. If this is not explicitly set, the default value used is 100.

If an attribute name is specified but is not present on a feature, then the sorting algorithm will consider that feature to have a null or zero value for that attribute.

An $O(n \log n)$ algorithm is used to do the sorting.

Assumptions

None.

Output Tags

The `SortFactory` supports the following output tag.

Tag	Description
<code>SORTED</code>	Applied to all features that leave this factory.

Example

The following example shows how the `SortFactory` can be used to order features by their areas:

```
FACTORY_DEF SHAPE SortFactory \
  INPUT FEATURE_TYPE * myarea @Area() \
  SORT_BY myarea NUMERIC \
  SORT_DIRECTION ASCENDING \
  OUTPUT SORTED FEATURE_TYPE * sizeRanking @Count(sizeRanking)
```

In this next example, the `SortFactory` sorts first by state name (in ascending order), and then by area, in descending order. (Note that the ordering for the area attribute is taken from the `SORT_DIRECTION` clause, while the ordering for the state name is given explicitly in the `SORT_BY` clause.)

```
FACTORY_DEF * SortFactory \
  INPUT FEATURE_TYPE * myarea @Area() \
  SORT_BY state_name ALPHA ASCENDING \
        area NUMERIC \
  SORT_DIRECTION DESCENDING \
  OUTPUT SORTED FEATURE_TYPE * sizeRanking @Count(sizeRanking)
```


SpatialFilterFactory

Note: The SpatialFilterFactory uses functionality from the GEOS library (<http://geos.refrations.net/>), which is distributed under the terms of the Free Software Foundation's LGPL license (<http://www.gnu.org/licenses/lgpl.html>). In order to comply with the terms of the LGPL, Safe Software Inc. has set up a website (<http://www.safe.com/foss>) to provide further information and to distribute source code as required.

Syntax

```

FACTORY_DEF <ReaderKeyword> SpatialFilterFactory
  [FACTORY_NAME <factory name>]
  [INPUT BASE FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]+
  [INPUT CANDIDATE FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]+
  PREDICATE [ ( INTERSECTS|
    DISJOINT|
    EQUALS|
    TOUCHES|
    CROSSES|
    OVERLAPS|
    CONTAINS|
    WITHIN|
    <DE-9IM string> ) ]+
  [BOOLEAN_OPERATOR ( AND|OR ) ]
  [USE_BOUNDING_BOX ( YES|NO|BASES_ONLY|CANDIDATES_ONLY ) ]
  [MULTIPLE_BASES ( YES|NO|BASES_FIRST ) ]
  [MERGE_BASE_ATTR ( YES|NO ) ]
  [BASE_ATTR_PREFIX <attribute name prefix> ]
  [PREDICATE_ATTR <attribute name> ]
  [CURVE_BOUNDARY_RULE ( ENDPOINTS_MOD2|ENDPOINTS_ALL ) ]
  [GROUP_BY [<attribute name>]* ]
  [OUTPUT ( PASSED|FAILED ) FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

Each SpatialFilterFactory accepts two kinds of features – *Bases* and *Candidates* – and tests topological (spatial) relationship predicates between them.

For example, if you have a group of features that you want to check against a polygon to isolate those that are inside, the polygon feature would be directed into the *BASE* clause and the group of features would be directed into the *CANDIDATE* clause. The *PREDICATE* clause would be *INSIDE*. Those features that are inside the polygon would come out the *PASSED* output clause, and those which were not inside would come out the *FAILED* output clause.

If there is more than one `BASE` feature, the behaviour depends on the value specified by `BOOLEAN_OPERATOR`; see below in the `Clauses` section for details. If there is more than one `PREDICATE` specified, each one is checked in the order they are specified until one is found that passes. All of the `PREDICATES` are checked on each `BASE` before moving to check the next `BASE`. If none of the `PREDICATES` is true for any of the `BASE` features, the `CANDIDATE` feature will come out the `FAILED` output clause.

The `SpatialFilterFactory` does not remove attributes or alter the geometry of the `CANDIDATE` features. It may, however, add new attributes.

If the `FME_GEOMETRY_HANDLING` directive is set to “yes” in the mapping file, arcs and ellipses are stroked prior to the testing of any relationship predicates; they are otherwise regarded as points located at their respective center points.

Assumptions

None.

Input Tags

The `SpatialFilterFactory` supports the following input tags.

Tag	Description
<code>BASE</code>	Features used as the criteria for the test.
<code>CANDIDATE</code>	Features to be tested against the given criteria.

Clauses

The `SpatialFilterFactory` supports several configuration clauses.

If `MULTIPLE_BASES` is set to `NO`, then only the first feature into the `BASE` input clause will be used. The rest will be ignored with a warning log message. If this is set to `YES`, then every base will be used but the `CANDIDATES` will be temporarily cached until all the features have passed through the factory. If this clause is set to `BASES_FIRST`, then all the `BASES` that arrive before the first `CANDIDATE` will be used, and any `BASE` features that arrive after the first `CANDIDATE` feature will be ignored with a warning log message. The default is setting is `YES`.

`BOOLEAN_OPERATOR` determines how the factory handles multiple `BASE` features. If `AND` is given, each `CANDIDATE` is checked against each `BASE` and must pass against each one. If `OR` is given, each `CANDIDATE` is checked against each `BASE`, sequentially, until one is found that passes. The default is `AND`.

If `MERGE_BASE_ATTR YES` is specified, then all *PASSED* features will have all the *BASE* attributes copied onto them, with the prefix specified by the `BASE_ATTR_PREFIX` clause. If the `BASE_ATTR_PREFIX` clause is not specified, or is an empty string, then the *BASE* attributes will not be given any prefix, and will only be copied to the candidate if those attributes do not already exist on the *CANDIDATE*. The default is setting is `NO`.

If the `PREDICATE_ATTR` is specified, then all *PASSED* features will have the name of the first predicate that passed added to the feature as the specified attribute.

The `USE_BOUNDING_BOX` predicate is used to indicate if the complete geometry of each feature should be used in the predicate evaluations or if the bounding boxes should be used instead. A value of `YES` means that the bounding boxes will be used for all *BASES* and *CANDIDATES*. A value of `NO` means that the complete geometry of both *BASES* and *CANDIDATES* will be used. A value of `BASES_ONLY` means that the bounding boxes of the *BASES* will be compared against the complete geometry of the *CANDIDATES*. A value of `CANDIDATES_ONLY` means that the complete geometry of the *BASES* will be compared against the bounding boxes of the *CANDIDATES*.

If `GROUP_BY` attributes are specified, *CANDIDATES* are only compared against *BASES* with the same values in these attributes.

The `CURVE_BOUNDARY_RULE` clause determines how the “boundary” of linear features is determined; see the *Spatial Predicate Background* section below for details.

The `PREDICATE` clause lists the spatial predicates that will be used between the *BASE* and *CANDIDATE* features. Note that if the spatial predicate is passed as an attribute, then the *CANDIDATE* feature must have the spatial relation predicate, not the *BASE*.

The definitions of each *PREDICATE* are given below. The *Spatial Predicate Background* section gives specific definitions of boundaries, exteriors, and interiors as they apply to specific feature types, and explains the concept of an intersection matrix.

Spatial Predicate Background

Each feature, whether it is a point, line, or polygon, etc. has a definition of an *INTERIOR*, *BOUNDARY* and *EXTERIOR*. The *EXTERIOR* is everything that is not on the *BOUNDARY* or the *INTERIOR*.

BOUNDARY, INTERIOR AND EXTERIOR

BOUNDARY	Points	Empty set.
	Lines	<p><code>CURVE_BOUNDARY_RULE ENDPOINTS_MOD2</code> The boundary is the set of all endpoints that occur an odd number of times. For a simple linear feature (i.e. not a multicurve), this means the boundary is comprised of the the start and end points, unless the line is closed (the start and end are the same point), in which case the boundary is the empty set. (This is the default if <code>CURVE_BOUNDARY_RULE</code> is unspecified.)</p> <p><code>CURVE_BOUNDARY_RULE ENDPOINTS_ALL</code> The boundary is the set of all endpoints, regardless of the number of times they occur in the geometry.</p>
	Polygons	The border of a polygon, including the border of the holes.
INTERIOR	Points	The point location.
	Lines	The entire line except its boundary, as determined above.
	Polygons	The inner surface of the polygon.

The comparison of two features produces a 3 x 3 matrix known as the Dimensionally Extended 9 Intersection Matrix (DE-9IM), shown below:

		candidate		
		interior	boundary	exterior
base	interior	χ_0	χ_1	χ_2
	boundary	χ_3	χ_4	χ_5
	exterior	χ_6	χ_7	χ_8

The value of each element of the matrix indicates the dimension of the geometry produced by intersecting the given parts of the two features. The dimension is one of the following:

- 1 there is no interaction
- 0 the intersection produces a point
- 1 the intersection produces a line
- 2 the intersection produces a surface

Each of the predicates can be defined in terms of what the intersection matrix of the two features must look like. For this, use a pattern matrix. Each element of the pattern matrix can be one of the following:

- The pattern matrix for the *disjoint* predicate is

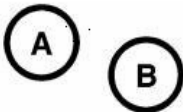

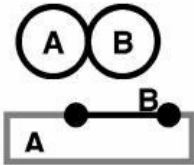
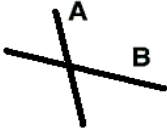
This means that neither feature's interior or boundary may interact with the other's interior or boundary.

Any intersection matrix can be represented as a 9-character string. To generate the string representation of a particular intersection matrix, take each element starting from the top-left, going right-to-left for each row.

Definitions of PREDICATE

Each of the supported predicates is described below, along with some associated examples and pattern matrices. Note that the examples are not

exhaustive: there may be entirely different situations where a given predicate is true. In the examples, the base is labelled "A" and the candidate is labelled "B."

Predicate	Example	Description	Pattern Matrix																											
INTERSECTS		The two features are not disjoint, as defined below.																												
DISJOINT		The boundaries and interiors do not intersect.	<table><tr><td>F</td><td>F</td><td>*</td></tr><tr><td>F</td><td>F</td><td>*</td></tr><tr><td>*</td><td>*</td><td>*</td></tr></table>	F	F	*	F	F	*	*	*	*																		
F	F	*																												
F	F	*																												
*	*	*																												
EQUALS		The features have the same boundary and the same interior.	<table><tr><td>T</td><td>*</td><td>F</td></tr><tr><td>*</td><td>*</td><td>F</td></tr><tr><td>F</td><td>F</td><td>*</td></tr></table>	T	*	F	*	*	F	F	F	*																		
T	*	F																												
*	*	F																												
F	F	*																												
TOUCHES		<p>The boundaries may intersect, or one boundary may intersect the other interior. The interiors do not touch.</p> <p>Undefined for point/point. or</p>	<table><tr><td>F</td><td>T</td><td>*</td></tr><tr><td>*</td><td>*</td><td>*</td></tr><tr><td>*</td><td>*</td><td>*</td></tr></table> <p>or</p> <table><tr><td>F</td><td>*</td><td>*</td></tr><tr><td>*</td><td>T</td><td>*</td></tr><tr><td>*</td><td>*</td><td>*</td></tr></table> <p>or</p> <table><tr><td>F</td><td>*</td><td>*</td></tr><tr><td>T</td><td>*</td><td>*</td></tr><tr><td>*</td><td>*</td><td>*</td></tr></table>	F	T	*	*	*	*	*	*	*	F	*	*	*	T	*	*	*	*	F	*	*	T	*	*	*	*	*
F	T	*																												
*	*	*																												
*	*	*																												
F	*	*																												
*	T	*																												
*	*	*																												
F	*	*																												
T	*	*																												
*	*	*																												
CROSSES		<p>The interiors intersect and the base's interior intersects the candidate's exterior. Or in the case of line/line, the intersection of the interiors forms a point.</p> <p>Undefined for point/point or area/area.</p>	<table><tr><td>T</td><td>*</td><td>T</td></tr><tr><td>*</td><td>*</td><td>*</td></tr><tr><td>*</td><td>*</td><td>*</td></tr></table> <p>or</p> <table><tr><td>0</td><td>*</td><td>*</td></tr><tr><td>*</td><td>*</td><td>*</td></tr><tr><td>*</td><td>*</td><td>*</td></tr></table> <p>(for two lines)</p>	T	*	T	*	*	*	*	*	*	0	*	*	*	*	*	*	*	*									
T	*	T																												
*	*	*																												
*	*	*																												
0	*	*																												
*	*	*																												
*	*	*																												

Example 1

```
# This example shows the use of the SpatialFilterFactory to
# select roads that are contained in regions defined by
# boundary.
```

```
FACTORY_DEF SHAPE SpatialFilterFactory      \
  INPUT BASE  FEATURE_TYPE boundary        \
  INPUT CANDIDATE FEATURE_TYPE roads       \
  OUTPUT PASSED FEATURE_TYPE *             \
  OUTPUT FAILED FEATURE_TYPE *            \
  USE_BOUNGING_BOX YES                    \
  MULTIPLE_BASES NO                      \
  MERGE_BASE_ATTR YES                    \
  PREDICATE   CONTAINS
```

Example 2

```
# This example shows the use of the SpatialFilterFactory to
# select roads that either cross boundaries or are contained by
# the regions defined by the boundaries. It is IMPORTANT to note
# that the PREDICATE ordering allows use to first test for roads
# that cross boundaries, and then test for roads that are
# contained in regions.
```

```
FACTORY_DEF SHAPE SpatialFilterFactory      \
  INPUT BASE  FEATURE_TYPE boundary        \
  INPUT CANDIDATE FEATURE_TYPE roads       \
  OUTPUT PASSED FEATURE_TYPE *             \
  OUTPUT FAILED FEATURE_TYPE *            \
  USE_BOUNGING_BOX YES                    \
  MULTIPLE_BASES NO                      \
  MERGE_BASE_ATTR YES                    \
  PREDICATE   OVERLAP CONTAINS
```

Example 3

```
# This example shows the use of the SpatialFilterFactory to
# select roads that either cross boundaries or are contained by
# the regions defined by the boundaries. It is IMPORTANT to
# note that the PREDICATE ordering allows use to first test for
# roads that cross boundaries, and then test for roads that are
# contained in regions.
    FACTORY_DEF SHAPE SpatialFilterFactory          \
        INPUT BASE  FEATURE_TYPE boundary          \
        INPUT CANDIDATE FEATURE_TYPE roads          \
        OUTPUT PASSED FEATURE_TYPE *                \
        OUTPUT FAILED FEATURE_TYPE *                \
        USE_BOUNDING_BOX YES                          \
        MULTIPLE_BASES NO                            \
        MERGE_BASE_ATTR YES                          \
        PREDICATE OVERLAP CONTAINS
```


SpatialRelationshipFactory

Note: The SpatialRelationshipFactory uses functionality from the GEOS library (<http://geos.refrations.net/>), which is distributed under the terms of the Free Software Foundation's LGPL license (<http://www.gnu.org/licenses/lgpl.html>). In order to comply with the terms of the LGPL, Safe Software Inc. has set up a website (<http://www.safe.com/foss>) to provide further information and to distribute source code as required.

Syntax

```

FACTORY_DEF <ReaderKeyword> SpatialFilterFactory
[FACTORY_NAME <factory name>]
[INPUT BASE FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]+
[INPUT CANDIDATE FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]+
PREDICATE [(INTERSECTS|
  EQUALS|
  TOUCHES|
  CROSSES|
  OVERLAPS|
  CONTAINS|
  WITHIN|
  <mask>|
  &<attribute name>
  @<function>)]+
[LIST_NAME <attribute name>]
[SUCCESS_ATTR <attribute name>]
[DIFFERING_ATTRIBUTES [<attribute name>]+]
[CALCULATE_CARDINALITY [(YES|NO)]]
[CURVE_BOUNDARY_RULE (ENDPOINTS_MOD2|ENDPOINTS_ALL)]
[GROUP_BY [<attribute name>]+]*
[OUTPUT TAGGED FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*

```

Overview

This factory accepts two kinds of features – *Bases* and *Candidates* – and tests topological (spatial) relationships between them. These relationships are also known as Egenhofer relationships. The factory tags (but does not otherwise change) the *BASE* features when these relationships are present.

All the *BASE* features are output via the *TAGGED* port, with a new list attribute appended. Each input *CANDIDATE* feature is compared against the *BASE* features, based on the specified spatial tests. When one of the comparisons is true, an entry will be added to the *BASE*'s list attribute as follows:

```
<LIST_NAME>{i}.de9im = [DE9IM string]
```

```

<LIST_NAME>{i}.pass{0} = [first true PREDICATE]
<LIST_NAME>{i}.pass{1} = [second true PREDICATE]
...
<LIST_NAME>{i}.pass{n} = [(n+1)th true PREDICATE]

```

All the attributes of the matching **CANDIDATE** feature will also be added to the list. Furthermore, each **BASE** receives the attributes of the **CANDIDATE(s)** that passed the relationship. When attributes are merged down from **CANDIDATE** to **BASE** features, existing attributes are not replaced. Therefore if the **CANDIDATES** and **BASES** have attributes with the same name, then the values will not be transferred down.

An example scenario for this factory would be if you had a group of states and rivers, and you wanted to determine which rivers were completely contained in which states. Then, the states would be directed into the **BASE** clause and the rivers would be directed into the **CANDIDATE** clause. The **PREDICATE** clause would be *CONTAINS*. Each **BASE** attribute would then receive a new list attribute, with one entry for each river that is contained in it.

Note that the **SpatialRelationshipFactory** does not remove attributes or alter the geometry of the **BASE** features; it simply adds attributes.

Also note that the **SpatialRelationshipFactory** must wait until all features have entered the factory before it can begin processing.

If the **FME_GEOMETRY_HANDLING** directive is set to yes in the mapping file, arcs and ellipses are stroked prior to the testing of any relationship predicates; they are otherwise regarded as points located at their respective center points.

Assumptions

None.

Input Tags

The **SpatialRelationshipFactory** supports the following input tags.

Tag	Description
BASE	Features used as the criteria for the test (i.e. the features which will be tagged with a list describing the relationships)..
CANDIDATE	Features to be tested against the given criteria (i.e., the features that may appear as entries in the lists)..

Clauses

The **SpatialRelationshipFactory** supports the following clauses:

T	0	*
*	2	*
F	*	*

If the `CALCULATE_CARDINALITY` clause is specified, then for each - match, the “cardinality” of this intersection is computed. Three attributes are added to the corresponding list entry: `card_point`, `card_line`, and `card_area`. These will each be given an integer value specifying how many points, lines, and areas result from the intersection. An area results from two overlapping areas. A line results, for instance, from two areas that touch, sharing a line segment, or two overlapping lines, or a line crossing an area. A point results from such interactions as two areas touching only at a vertex, two crossing lines, or a point contained in a line or in an area.

The `CURVE_BOUNDARY_RULE` clause determines how the “boundary” of linear features is determined; see the *Spatial Predicate Background* section in the `SpatialFilterFactory` chapter for details.

The `GROUP_BY` clause is used to indicate that only `BASES` and `CANDIDATES` which have the same value for certain attributes should be compared. That is, if `GROUP_BY` attributes are specified, candidates are only compared to bases that have the same values in these attributes.

Output Tags

The `SpatialRelationshipFactory` supports the following output tags.

Tag	Description
TAGGED	These will be the <code>BASE</code> features, with the new attributes added. One list entry will be made for each of the candidates that has at least one matching predicate.

Example 1

```
# This example shows the use of the SpatialRelationshipFactory
# to
# find all the relationships amongst a set of roads. The
# relationships will be described in the relationships
# attribute
# that will be appended each base. The number of candidates
# that
# have at least one relationship in common with the base will
# be
# stored in num_related_candidates. Furthermore, candidates
# must
# have a different value from bases in road_id, otherwise they
# will
# not be compared.

FACTORY_DEF SHAPE SpatialRelationshipFactory \
  INPUT BASE FEATURE_TYPE roads \
  INPUT CANDIDATE FEATURE_TYPE roads \
  PREDICATE ALL \
  LIST_NAME relationships \
  SUCCESS_ATTR num_related_candidates \
  DIFFERING_ATTRIBUTES road_id \
  OUTPUT TAGGED FEATURE_TYPE *
```

Example 2

```
# This example shows the use of the SpatialRelationshipFactory
# to
# find relationships between states and lakes/ponds, where only
```



```

bases
# and candidates with the same timezone will be compared. Each
# candidate specifies which relationships to check (in a variable
# called predicates_to_check).

FACTORY_DEF SHAPE SpatialRelationshipFactory \
    INPUT BASE FEATURE_TYPE states \
    INPUT CANDIDATE FEATURE_TYPE lakes \
    INPUT CANDIDATE FEATURE_TYPE ponds \
    PREDICATE &predicates_to_check \
    LIST_NAME relationships \
    SUCCESS_ATTR num_related_candidates \
    GROUP_BY timezone \
    OUTPUT TAGGED FEATURE_TYPE *

```

Example 3

```

# This example shows the use of the SpatialRelationshipFactory
# to
# find several types of relationships at once, through the use
# of
# multiple predicates.

FACTORY_DEF SHAPE SpatialRelationshipFactory \
    INPUT BASE FEATURE_TYPE states \
    INPUT CANDIDATE FEATURE_TYPE lakes \
    PREDICATE OVERLAP CONTAINS COVERS \
    LIST_NAME relationships \
    SUCCESS_ATTR num_related_candidates \
    OUTPUT TAGGED FEATURE_TYPE *

```


SpikeRemoverFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> SpikeRemoverFactory           \
[FACTORY_NAME <factory name>]                             \
[INPUT FEATURE_TYPE <factory type>                         \
  [<attribute name> <attribute value>]*                     \
  [<feature function>]*]**                                  \
[SPIKE_ANGLE <maximum spike angle>]                       \
[SPIKE_LENGTH <maximum spike length>]                     \
[OUTPUT CHANGED * FEATURE_TYPE <feature type>            \
  [<attribute name> <attribute value>]*                     \
  [<feature function>]*]**                                  \
[OUTPUT FLAGGED * FEATURE_TYPE <feature type>            \
  [<attribute name> <attribute value>]*                     \
  [<feature function>]*]**                                  \
[OUTPUT UNCHANGED * FEATURE_TYPE <feature type>          \
  [<attribute name> <attribute value>]*                     \
  [<feature function>]*]**

```

Overview

Cleans up feature geometries by removing spikes in 2D.

The factory looks at every pair of line segments made up of three consecutive distinct points. If the angle between two line segments is less than or equal to the <maximum spike angle>, then the middle point is a spike and is removed.

If the <maximum spike length> is specified, then the factory will skip line segments longer than this length. Otherwise all line segments are considered.

If the geometry of a feature is a path, the factory removes spikes between consecutive path segments as well. For a polygon or donut, if the start/end point is a spike, then it is also removed. The end result is still a polygon/donut. Any polygons, donuts, paths or lines that are part of a collection of geometry will also be processed.

The factory will also remove any duplicate points.

The factory is not effective when the line contains many deviations other than spikes. In such cases, it is recommended to first clean up the features using the LineGeneralizer transformer with Douglas-Poiker algorithm.

Assumptions

None.

Output Tags

The `SpikeRemoverFactory` supports the following output tag.

Tag	Description
CHANGED	Features that are cleaned up.
FLAGGED	Duplicate points and spikes.
UNCHANGED	Untouched features.

Examples

This example shows the factory being used to remove spikes that have angle less than or equal to 5 degrees and length smaller than or equal to 2.

```
FACFACTORY_DEF * SpikeRemoverFactory          \  
  FACTORY_NAME SpikeRemover                    \  
  INPUT  FEATURE_TYPE LineGeneralizer_OUTPUT_1  \  
  SPIKE_ANGLE 5                                \  
  SPIKE_LENGTH 2                               \  
  OUTPUT CHANGED * FEATURE_TYPE SpikeRemover_OUTPUT
```

SurfaceModelFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```

FACTORY_DEF <ReaderKeyword> SurfaceModelFactory           \
[FACTORY_NAME <factory name>]                               \
[TOLERANCE <num>]                                           \
[SAMPLE_SPACING <x_spacing> <y_spacing>]                   \
[NODATA_VALUE <float>]                                       \
[SAMPLE_DIMENSIONS <number of rows> <number of columns>] \
[CONTOUR_INTERVAL <float>]                                   \
[CONTOUR_DIMENSION (2|3)]                                   \
[INTERPOLATION_TYPE (AUTO|PLANER|CONSTANT)]                 \
[DRAPE_TYPE (VERTEX|MODEL)]                                  \
[SURFACE_BOUNDING_BOX <minX> <minY> <maxX> <maxY>]         \
[QUERY_BOUNDING_BOX <minX> <minY> <maxX> <maxY>]         \
[SURFACE_FILE_BASE_NAME <path> (APPEND|OVERWRITE)]         \
[INPUT (3D_LINES|BREAKLINES|DRAPE_FEATURES|POINTS)]        \
FEATURE_TYPE <feature type>                                  \
  [<attribute name> <attribute value>]*                       \
  [<feature function>]* ]*                                     \
[OUTPUT (CONTOURS|DEM_POINTS|DEM_GRID|DRAPED_FEATURE|      \
TIN_EDGES|
  TRIANGLES|VERTEX_POINTS|VERTEX_POINTS_WITH_ATTRS|
  VORONOI_DIAGRAM|VORONOI_DIAGRAM_WITH_ATTRS)                \
FEATURE_TYPE <feature type>                                  \
  [<attribute name> <attribute value>]*                       \
  [<feature function>]* ]*

```

Overview

This factory takes three-dimensional features and uses them to define a surface model. The underlying surface model is a Delaunay Triangulation. The model is built using the **TOLERANCE** and the 3D features that are input to the factory through the **3D_LINES**, **POINTS**, and **BREAKLINES** clauses.

The factory is useful for converting from one surface representation to another and for adding *z* values to features. This is useful for cases where you may have two-dimensional features and elevation data either in the form of a DEM or a collection of other 3D features, and would like to add the third dimension to the features.

The **TOLERANCE** clause is used to specify the accuracy of the surface model being constructed. The larger the value specified for tolerance, the greater the thinning of the input data.

The **INTERPOLATION_TYPE** clause is used if the output of the model is requested as **DEM_POINTS** or if draping features are input to the model. If **CONSTANT** is specified, the *z* value of each output point is set to that of the closest vertex of

the underlying model. If `PLANAR` is specified, then planar interpolation is used to determine the `z` value for each output point. If a point is not over a surface triangle and `PLANAR` is specified, the output `z` value will be set to `NAN`. If `AUTO` is specified, the algorithm will attempt to give the best possible result for each point. In the current version of `AUTO`, the planar method is used if the point is over a surface triangle and the constant method is used otherwise.

The `DRAPE_TYPE` clause is used if any features are accepted by the model through the `DRAPE_FEATURES` input tag. Any features that are input to the factory through this clause will be returned with their attribution unchanged; however, they will now be 3D, having their `z` value set according to the value of `INTERPOLATION_TYPE`. If `DRAPE_TYPE` is `VERTEX`, the feature will be returned with the same number of vertices that it was input with, and with `z` values added to each vertex on the feature. If the `DRAPE_TYPE` is `MODEL`, the feature will have other coordinates added to it, forcing it to exactly follow the underlying surface model.

The `NODATA_VALUE` clause is used to fill the raster cell values that fall outside of the surface's bounding box. If this is not specified, `NAN` values will be used instead. Use of this clause is highly recommended in order to produce consistent raster data with nodata values rather than `NAN` values.

The `SURFACE_BOUNDING_BOX` clause defines the extent of the model. When building large surface models, it is strongly recommended that you specify the total size of the model. If this is not specified, then the surface is constructed using heuristics based on the first 50,000 vertices entered.

The `QUERY_BOUNDING_BOX` constrains the output from the model to be that which overlaps with the `QUERY_BOUNDING_BOX`. If the `QUERY_BOUNDING_BOX` is not specified, then the `QUERY_BOUNDING_BOX` defaults to the extent of the entire underlying surface model.

The `SURFACE_FILE_BASE_NAME` associates the surface with a file. If the mode is `OVERWRITE`, the newly generated surface will be written to the file. The original contents of the file (if any) will be lost. If the mode is `APPEND`, the existing surface from the file will be combined with the newly generated surface (if any). The file will then be replaced with the combination.

If the output tag `DEM_POINTS` is specified, then `SAMPLE_SPACING` is used to specify spacing of the output `DEM_POINTS`. If `SAMPLE_SPACING` is not specified, then 1 is used.

If the output tag `DEM_GRID` is specified, then `SAMPLE_SPACING` is used to specify spacing of the output points that will be used to construct a grid. Alternatively, `SAMPLE_ROWS_AND_COLUMNS` can be specified to establish the size of the output `DEM_GRID`. Only one or the other may be specified.

If the output tag `CONTOURS` is specified, then `CONTOUR_INTERVAL` is used to specify the elevation separation of the output `CONTOURS`. The `CONTOUR_INTERVAL` has a default value of 50.0. The value specified for `CONTOUR_DIMENSION` is also used to specify whether the output contours are to be 2D or 3D. The default for `CONTOUR_DIMENSION` is 2.

See below for a description of the other types of output clauses that are supported.

Assumptions

None.

Input Tags

Tag	Description
3D_LINES	The vertices of the 3D line are inserted into the model to define the surface. They are not used to define breaklines but rather just provide 3D data for the model from their vertices.
BREAKLINES	These features are put into the model as breaklines such that triangle edges will always be along the line of the feature.
DRAPE_FEATURES	Features input through this tag are output via the <code>DRAPED</code> tag with their <code>z</code> value set to the values from the surface model. It is not a requirement that the input feature be 2D. If the input feature is 3D then the <code>z</code> dimension is simply replaced with that from the model.
POINTS	The vertices from the 3D points are used to define the surface model. Each vertex of the input feature is used to define the underlying model. This tag also has the ability to accept FME grid features. Points are extracted from a grid, and the vertices of these 3D points are used to define the surface model.

Output Tags

Tag	Description
CONTOURS	If specified, then the model is output as a set of contours through this clause. Each feature is 2D or 3D depending on the value specified for <code>CONTOUR_DIMENSION</code> . Each output feature also has an attribute <code>SurfaceModel.elevation</code> that holds the elevation of the contour. If the output feature is 3D, then this value is the same as the <code>z</code> coordinate on each feature.
DEM_POINTS	Returns the model as a series of evenly spaced 3D points based on the sampling rate specified.

Tag	Description
DEM_GRID	Returns the model as a single grid feature consisting of evenly spaced 3D points arranged by rows and columns. The spacing between rows and columns is based on the sample spacing specified. Alternatively, the number of rows and columns can be specified to establish the size of the grid, instead of the sample spacing. Only one or the other may be specified.
DRAPED_FEATURES	Features input via the <code>DRAPED_FEATURES</code> tag are output here with the z dimension set to values from the model.
TIN_EDGES	Outputs all the edges which define the underlying Delaunay Triangulation. Each feature output on this clause has the attributes <code>SurfaceModel.vertex1_id</code> and <code>SurfaceModel.vertex2_id</code> , which identifies the ID of the vertex to which it was connected.
TRIANGLES	Returns the Triangles that define the Delaunay Triangulation. Each feature output on this tag has the attributes <code>SurfaceModel.vertex1_id</code> , <code>SurfaceModel.vertex2_id</code> , and <code>SurfaceModel.vertex3_id</code> which identify the ID's of the vertices. <code>SurfaceModel.slope</code> which is the slope of the triangles in degrees. <code>SurfaceModel.aspect</code> which is aspect angle in degrees measured from the x-axis in a counter-clockwise direction. <code>SurfaceModel.percentageSlope</code> which is the percentage slope.
VERTEX_POINTS	Outputs each vertex of the underlying surface model triangulation. Each vertex is tagged with the unique identifier <code>SurfaceModel.vertex_id</code> .
VERTEX_POINTS_WITH_ATTRS	Outputs each vertex of the underlying surface model triangulation. Each triangle is tagged with the unique identifier <code>SurfaceModel.vertex_id</code> . This clause differs from the <code>VERTEX_POINTS</code> output clause as it only outputs those point features that were accepted through the <code>POINTS</code> input clause (that is, those points that have attributes with the attributes maintained). Any points that are introduced into the model through breaklines processing will not be output on this clause.
VORONOI_DIAGRAM	Outputs the dual of the Delaunay Triangulation called the Voronoi Diagram. For each vertex of the Delaunay Triangulation, a Voronoi polygon feature is returned. The Voronoi polygon has the attributes <code>SurfaceModel.vertex_id</code> identifying the id that it represents and <code>SurfaceModel.elevation</code> which gives the elevation of the point and hence the voronoi polygon.

Tag	Description
VORONOI_DIAGRAM_WITH_ATTRS	Outputs the dual of the Delaunay Triangulation called the Voronoi Diagram. For each vertex of the Delaunay Triangulation, a Voronoi polygon feature is returned. The Voronoi polygon has the attributes <code>SurfaceModel.vertex_id</code> identifying the ID that it represents, and <code>SurfaceModel.elevation</code> which gives the elevation of the point and hence the Voronoi polygon. This clause differs from the <code>VORONOI_DIAGRAM</code> clause above as it only outputs Voronoi polygons for those point features that were accepted through the <code>POINTS</code> input clause (that is, those points that have attributes with the attributes maintained). Any points that are introduced into the model through break-lines processing will not have Voronoi polygons output.

Example

In the example below, all features with the feature type `ELEVATION` are used to define a surface based on a 50-unit tolerance. As directed by the mapping file below, the surface is then output in four different representations:

- a Voronoi Diagram,
- the Delaunay Triangles that define the surface,
- the triangle vertices that represent the surface, and
- all the triangle edges.

```
FACTORY_DEF * SurfaceModelFactory \
  FACTORY_NAME SURFACEMODELLER \
  INPUT POINTS FEATURE_TYPE ELEVATION \
  TOLERANCE 50 \
  OUTPUT VORONOI_DIAGRAM FEATURE_TYPE \
SURFACEMODELLER_VORONOI_DIAGRAM \
  OUTPUT TRIANGLES FEATURE_TYPE SURFACEMODELLER_TRIANGLES \
  OUTPUT VERTEX_POINTS FEATURE_TYPE \
SURFACEMODELLER_VERTEX_POINTS \
  OUTPUT TIN_EDGES FEATURE_TYPE SURFACEMODELLER_TIN_EDGES
```

TeeFactory

Syntax

```
FACTORY_DEF <ReaderKeyword> TeeFactory
  [FACTORY_NAME <factory name>]
  [NUMBER_OF_COPIES <copy count>]
  [COPY_NUMBER_ATTRIBUTE <attribute name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [OUTPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
```

Overview

This factory implements a T junction in the factory pipeline. It replicates each feature that matches the input specification, and outputs a copy of the feature for each **OUTPUT** clause specified. The output clauses may change the feature's type, or otherwise alter the feature, as it goes through. If the **INPUT** clause is not specified, then every feature will be input into the factory. This factory is useful when copies of features need to be generated for further processing by other factories, or for output to different layers or themes in the output data set.

Note that if no **OUTPUT** clauses are specified, the input features will be deleted.

The **NUMBER_OF_COPIES** clause may be used to specify how many copies of each input feature are written to each **OUTPUT** clause. If this is not specified, a value of 1 is assumed.

When more than a single copy of the feature is written to each **OUTPUT** clause, the **COPY_NUMBER_ATTRIBUTE** clause may be specified to name an attribute which will be defined on the output features to specify which of the copies they represent. If specified, the first feature output will have a value of "0" for the specified attribute, and the value will increment for subsequent copies.

Assumptions

None.

Output Tags

Since each output clause is applied to all input features, this factory has no output tags.

Example

The following example uses the `TeeFactory` to make three copies of each `Road::TRIM` feature flowing through the pipeline. The first copy is output by the first `OUTPUT` clause, and flows through the pipeline unchanged as this clause does nothing to it. The second copy has its feature type changed to `Douglased`, and has its geometry generalized using the Douglas algorithm with a tolerance of 20 meters. The third copy has its feature type changed to `deveaued` and has its geometry generalized using the Deveau algorithm with a tolerance of 10 meters, 3 wedges, and a 100-degree blunting angle.

```

FACTORY_DEF SAIF TeeFactory                                \
  INPUT FEATURE_TYPE Road::TRIM                            \
  OUTPUT FEATURE_TYPE *                                     \
  OUTPUT FEATURE_TYPE Douglased                            \
    @Generalize(Douglas,20)                                 \
  OUTPUT FEATURE_TYPE deveaued                             \
    @Generalize(Deveau,10,3,100)                           \

```

TestFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> TestFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  [TEST <value> <operator> <value>]+
  [STRING_TEST <value> <operator> <value>]+
  [BOOLEAN_OPERATOR (AND|OR)]
  [OUTPUT (PASSED|FAILED) FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory takes a feature and performs the tests defined by the **TEST** clauses. If the result of the tests is true, then the feature will be output using the **PASSED** tag. If the result is false, then the feature will be output by the **FAILED** tag.

If the **PASSED** clause is not specified, then features that pass the test will be destroyed. If the **FAILED** clause is not specified, then features that fail the test will be destroyed. Either **PASSED** or **FAILED** must be present.

The <operator> is one of <, >, =, ==, !=, >=, or <=.

The <value> may be a literal constant, an attribute name preceded by the value-of operator (&), or an attribute value function. If it is an attribute value function, the function will be executed on the current feature and the result will be used for the test.

Example **TEST** clauses include:

- **TEST** @Area() < 100
- **TEST** &numLanes > 2
- **STRING_TEST** "Joe" = "Jerry"

At run-time the **TestFactory** decides to invoke numeric comparisons or string comparisons, as greater than and less than, that have different meanings depending on the type of operands. If both arguments may be converted to numbers, then numeric comparisons will be used; otherwise, string comparisons will be used.

If multiple **TEST** or **STRING_TEST** clauses are present, the test will only pass if all of the **TEST**s are true, unless the **BOOLEAN_OPERATOR OR** clause is specified. (By default, the Boolean clause used between multiple tests is **AND**.) The **STRING_TEST** clause forces string comparison of the equation, and the

TEST clause (which is the default) indicates that a basic evaluation of the tests is performed.

Note that you cannot directly test for the value of TEST. However, you can perform the test by following the steps below:

- 1. In Workbench, use an AttributeCreator to add a new attribute, and set its value to TEST.
 - Use the value of the new attribute to test against.

Assumptions

None.

Output Tags

The TestFactory supports the following output tags.

Tag	Description
PASSED	Applied to features where the TEST clause is true.
FAILED	Applied to features where the TEST clause is false.

Example

The following example shows how the TestFactory can be used to perform area generalization on area features.

The first factory defined below deletes all area features with an area less than 1000 square ground units:

```
FACTORY_DEF SHAPE TestFactory \
  INPUT FEATURE_TYPE * fme_geometry fme_polygon \
  TEST @Area() < 1000 \
  OUTPUT FAILED FEATURE_TYPE *
```

The second factory converts all area features with an area of less than 10000 square ground units to a point and passes the rest on to the third factory.

```
FACTORY_DEF SHAPE TestFactory \
  INPUT FEATURE_TYPE * fme_geometry fme_polygon \
  TEST @Area() < 10000 \
  OUTPUT PASSED FEATURE_TYPE Point @ConvertToPoint() \
  OUTPUT FAILED FEATURE_TYPE *
```

The final factory converts all remaining area features with a circularity of less than 0.25 to a line and generalizes those with a circularity greater than or equal to 0.25.

```
FACTORY_DEF SHAPE TestFactory                                \  
  INPUT FEATURE_TYPE * fme_geometry fme_polygon            \  
  TEST @Circularity() < 0.25                                \  
  OUTPUT PASSED FEATURE_TYPE Line @ConvertToLine(100)      \  
  OUTPUT FAILED FEATURE_TYPE Polygon @Generalize(Douglas, 100)
```

TextStrokerFactory

Note: This factory uses software from the FreeType Project © 1996-2000 by David Turner, Robert Wilhelm, and Werner Lemberg.

Syntax

```

FACTORY_DEF <ReaderKeyword> TextStrokerFactory
[FACTORY_NAME <factory name>]
[INPUT FEATURE_TYPE <feature type>
  [<attribute name> <attribute Value>]*
  [<feature function>]*]*
[TEXT_STRING_ATTR <attribute name>]
[TEXT_FONT [<truetype font name>]+]
[TEXT_FONT_WIDTH_MULTIPLIER [<attribute name>]+]
[TEXT_FONT_HEIGHT [<attribute name>]+]
[TEXT_PADDING [padding]+]
[TEXT_ROTATION [<attribute name>]+]
[BUILD_POLYGONS [<attribute name>]+]
[TEXT_FONT_PATH <font path>]
[OUTPUT (STROKED|UNTOUCHED)
  FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*

```

Overview

This factory is used to stroke text into polygons based on TrueType font definitions from an input feature and text string. The factory accepts features with point geometry and replaces the geometry with an aggregate of polygons that visually represent the text. Any feature received that does not have a `POINT` geometry, or has a text string consisting of zero characters or all space characters, is output by the `UNTOUCHED` output tag.

The resulting text has several additional attributes taken from the input feature and other values passed into the factory. The text height, width and padding are given in ground units and determine the size and spacing of the text. The rotation is given in degrees. The font name defaults to Arial, but can be set to any valid TrueType font.

On non-Windows systems, the path to the font file can be specified in the mapping file using the `TEXT_FONT_PATH` keyword. The font name supplied must then be the filename for the requested font (for example, `bkant.ttf`).

Assumptions

The features to be stroked are assumed to enter the factory with all required attributes (`text_font`, `text_padding`, `build_polygons`) set. If any feature

does not contain values for these attributes, and the values are not supplied, then default values will be used.

The font supplied is assumed to be a valid TrueType font installed on the system. If an attribute is used for the font name instead of a literal, the attribute must contain the font's style information. The resulting string must have the following format (entries in square brackets are optional):

```
<fontname>,[BOLD],[ITALIC],[<character set index within font>]
```

The origin (justification) of the resulting text outline is always the lower left-hand corner.

Clauses

The `TextStrokerFactory` requires that a number of clauses be specified.

Clauses	Description	Optional
TEXT_STRING <field name>	Specifies the name of the feature attribute that holds the text string. Default: <code>fme_text_string</code> Example 2: <code>TEXT_STRING_ATTR @Value(Text)</code>	Yes
TEXT_FONT <field name>	The name of the font in addition to its style and character set. On Windows systems, this must be a valid TrueType font (note that this is not necessarily the name of the font file). Default: <code>Arial</code> Example: <code>TEXT_FONT Georgia,,ITALIC,</code> Example: <code>TEXT_FONT Garamond,BOLD,ITALIC,2</code> Example 2: <code>TEXT_FONT @Value(Font)</code>	No
TEXT_FONT_HEIGHT <value>	Specifies the name of the feature attribute that holds the font height in ground units. A value of zero indicates that the font height is equal to the font width. Default: <code>fme_text_size</code> Example: <code>TEXT_FONT_HEIGHT 1</code>	Yes
TEXT_FONT_WIDTH_MULTIPLIER <value>	Specifies the name of the feature attribute that holds the font width in ground units. A value of zero indicates that the font width is equal to the font height. Default: <code>1.0</code> Example: <code>TEXT_FONT_WIDTH_MULTIPLIER 1.5</code>	Yes

Clauses	Description	Optional
TEXT_PADDING <value>	Specifies the text padding (extra spacing between characters) in ground units. Default: 0 Example: TEXT_PADDING 0	No
TEXT_ROTATION <value>	Specifies the name of the feature attribute that holds the text rotation, measured in degrees counterclockwise from horizontal. Default: fme_text_rotation Example: TEXT_ROTATION 45 Example 2: TEXT_ROTATION @Value(Rot)	Yes
BUILD_POLYGONS <value>	Specifies whether the stroked text should be output as polygons or lines. Default: Polygon features Example: BUILD_POLYGONS YES	No

Output Tags

The TextStrokerFactory supports the following output tags.

Tag	Description
STROKED	This tag outputs aggregate polygon features constructed by the factory.
UNTOUCHED	This tag outputs unprocessed features.

Example

The following example constructs polygon features from point features. The input file has the following definition:

ID	X	Y	Text	Font	Padding	Rotation	Text Height
1	0	0	The quick brown	Arial	0	0	1
2	10	10	fox	Garamond	0	25	2
3	30	30	jumps	Centaur	1	90	2
4	100	100	over the	Broadway	0	300	3
5	500	500	lazy	Forestry	0	720	10
6	600	600	dog	Arial	0	360	30

Using this file as input for the `TextStrokerFactory` and setting the corresponding attributes in the transformer will output six aggregate features describing the stroked outlines of the six input records.

Factory Definition

```

FACTORY_DEF * TextStrokerFactory      \
    INPUT FEATURE_TYPE points         \
    TEXT_STRING_ATTR &TEXT            \
    TEXT_FONT &FONT                   \
    TEXT_FONT_HEIGHT &FONT_HEIGHT     \
    TEXT_FONT_WIDTH_MULTIPLIER &FONT_WIDTH_MULT \
    TEXT_PADDING &PADDING             \
    TEXT_ROTATION &ROTATION           \
    BUILD_POLYGONS &BUILD_POLYS      \
    OUTPUT STROKED FEATURE_TYPE stroked

```

TilingFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> TilingFactory
  [FACTORY_NAME <factory name>]
  [INPUT FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function> ]]*
  [GROUP_BY [<attribute name>+]*
  TILE_WIDTH <width>
  TILE_HEIGHT <height>
  [MINIMUM_MEMBERS <members>]
  [OUTPUT TILED FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function> ]]*

```

Overview

This factory creates aggregates from its input features, grouping them into tiles covering approximately the width and height specified in the factory definition.

This factory is useful to reduce the data volume of *wallpaper* types of features that have **no individual attributes**. The resulting aggregates, or tiles, can be output to a system using many fewer records than if each feature was output by itself. For systems that support aggregates, or multi-part features, this can result in substantial performance improvements and greatly decrease storage requirements.

The factory uses the tile <width> and <height> which are measured in ground units to divide the input space into cells. The center of the bounding box of each input feature is used to determine the cell for the feature. Once all input features have been read in, an aggregate feature is created from all features in each cell. If linear features are input, they will have pseudo nodes removed within their cells to further reduce the number of separate entities. No such reduction is done to any polygons or donuts that enter.

If the `MINIMUM_MEMBERS` clause is specified, then tiles that have fewer than this number of features in them before pseudo-node removal will be merged with a vertically neighbouring tile to increase the number of members.

Features that leave the factory have only the group-by attributes present on them. Any other feature attributes are lost.

If the `FME_GEOMETRY_HANDLING` directive is set to “yes” in the mapping file, the geometric bounding box of arcs and ellipses are used to group the features into tiles. Otherwise, arcs and ellipses are demoted to center points prior to grouping the features into tiles.

Assumptions

The factory does not clip any input features. This means that a feature that spans multiple cells causes one tile to overlap with its neighbours. It is assumed that the input data does not contain features considerably larger than the tile width and height.

Output Tags

The `TilingFactory` supports the following output tag.

Tag	Description
TILED	The tiles formed from the input features.

Example

In this example, the `TilingFactory` is used to tile data read from a Design file. The input data is measured in latitude/longitude so each tile will be approximately 0.25 degree wide and 0.125 degree high. Each tile also contains at least 50 original features, which means that some tiles are joined with their vertical neighbors to ensure they have that much data in them.

The input data is also separated by the value of the `igds_type` attribute: in this case, features whose type was `igds_shape` are not tiled with features whose type is `igds_line`.

```

FACTORY_DEF * TilingFactory                                \
  INPUT FEATURE_TYPE * igds_type igds_shape                \
  INPUT FEATURE_TYPE * igds_type igds_line                 \
  GROUP_BY igds_type                                       \
  TILE_WIDTH 0.25                                          \
  TILE_HEIGHT 0.125                                       \
  MINIMUM_MEMBERS 50                                      \
  OUTPUT TILED FEATURE_TYPE *
```

TopologyFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> TopologyFactory
[FACTORY_NAME <factory name>]
[INPUT FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]+
[GROUP_BY [<field name>]+]*
[MAX_COORDS <integer>]
[ARC_NUMBER_ATTR <field name>]
[FROM_NODE_ATTR <field name>]
[TO_NODE_ATTR <field name>]
[RIGHT_POLY_ATTR <field name>]
[LEFT_POLY_ATTR <field name>]
[LEFT_EDGE_ATTR <field name>]
[RIGHT_EDGE_ATTR <field name>]
[NODES_ATTR <field name>]
[PROPAGATE_ATTRS (yes|no)]
[POLYGONS_ATTR <field name>]
[POLYGON_NUMBER_ATTR <field name>]
[ARCS_ATTR <field name>]
[PERIMETER_ATTR <field name>]
[AREA_ATTR <field name>]
[NODE_NUMBER_ATTR <field name>]
[ANGLE_PREFIX_ATTR <field name>]
[UNIVERSE_GEOMETRY [(yes|no)]]
[IGNORE_NODE_HEIGHTS [(yes|no)]]
[OVERLAPPING_DATA_PREFIX_ATTR <field name>]
[START_EDGES_ONLY [(yes|no)]]
[ALLOW_CYCLES [(yes|no)]]
[ASSUME_CLEAN_DATA [(yes|no)]]
[OUTPUT (LINE|POLYGON|NODE|UNIVERSE_POLYGON|ILLEGAL_GEOM)
  FEATURE_TYPE <feature type>
  [<attribute name> <attribute value>]*
  [<feature function>]*]*

```

Overview

This factory takes point, linear and/or polygonal features, pushes them into a topology model, and outputs them with all of their topology information. Topologically significant nodes, lines, and polygons are computed using all input features. The factory does not assume that all input data is clean and noded properly, unless **ASSUME_CLEAN_DATA** is set to yes. If **ASSUME_CLEAN_DATA** is not specified or set to no, the factory takes any data and constructs the resulting topology after computing any intersections present in the input data. Otherwise, the input is assumed to be noded properly and clean, thus, intersection work is skipped.

The topology can be constructed in 3D, if requested. Constructing a topology in 3D would mean that line segments would only be topologically linked if they shared the same Z value at the point they intersected. Constructing a topology in 2D would mean that all intersecting segments would be topologically connected to each other at a common node, regardless of their respective Z values. Consider, for example, a situation where two lines (which crossed) represented roads, where one road was an overpass above the other road. Suppose these two lines had differing elevations. If you constructed the topology in 3D, these two roads would not be linked to the same node where they crossed. If the topology was constructed in 2D, these lines would both link to a common node, which would be present at the location where they crossed. In either the 2D or 3D case, the full dimensionality of the input is preserved in the output – 3D features are never converted to 2D. The 2D or 3D choice only indicates how the topology is created; it does not affect the dimension of the features output.

By default, the `UNIVERSE_POLYGON` does not have any geometry when it is returned. It does have a list of the arcs which form the universe polygon. If the `UNIVERSE_GEOMETRY` directive is set to *yes*, then the universe polygon will also have a polygonal geometry. If any input polygons are 3D, then the universe polygon will be 3D; otherwise it will be 2D.

All output lines that form the border of a polygon and are collinear with input lines will have the attributes from the input lines merged onto the output border lines. No original attributes are overwritten.

The input features may be *grouped* into separate topology sets based on attribute values. All attributes are carried across from the `INPUT` features to the output features.

The factory has the following clauses that specify the attribute names to use for assigning topology information. The clauses are as follows:

Clause	Description
<code>MAX_COORDS</code>	The number indicates the maximum length to output any line. If any line contains more than this number of coordinates, it will be broken into pieces which are output separately, each with their own line IDs, and correctly noded.
<code>UNIVERSE_GEOMETRY</code>	If this is <i>yes</i> , the universe polygon returned will have complete geometry on it. If it is <i>no</i> , the universe polygon will take less time to create because it will have no geometry, but will have links to the arcs that form the boundary of the polygon. The default is <i>no</i> .
<code>IGNORE_NODE_HEIGHTS</code>	If this is <i>yes</i> , the topology will be computed in 2D. If it is <i>no</i> , the topology will be computed in 3D. The default is <i>yes</i> .

Clause	Description
ARC_NUMBER_ATTR	The name of the attribute field used on each output linear element to contain its unique identifier. Each line which is output from the model has a unique identifier, numbered increasing from 1. (This attribute will appear on output <code>LINES</code> only.)
FROM_NODE_ATTR	The attribute into which the ID of the starting node is placed. The nodes are uniquely numbered increasing from 1. (This attribute will appear on output <code>LINES</code> only.)
TO_NODE_ATTR	The attribute into which the ID of the end node is placed. The nodes are uniquely numbered increasing from 1. (This attribute will appear on output <code>LINES</code> only.)
RIGHT_POLY_ATTR	The field into which the ID of the polygon on the right side is placed. (If there are multiple such polygons, only one of their IDs will be noted.) The right side is defined to be that polygon which is on the right of the line when travelling from the <code>FROM_NODE</code> to the <code>TO_NODE</code> along the arc. The <code>UNIVERSE_POLYGON</code> is given an ID of 0. (This attribute will appear on output <code>LINES</code> only that were originally input as polygons. Any output <code>LINES</code> originally input as lines will not have this attribute when output.)
LEFT_POLY_ATTR	The field into which the ID of the polygon on the left side is placed. (If there are multiple such polygons, only one of their IDs will be noted.) The left side is defined to be that polygon which is on the left of the line when travelling from the <code>FROM_NODE</code> to the <code>TO_NODE</code> along the arc. The <code>UNIVERSE_POLYGON</code> is given an ID of 0. (This attribute will appear on output <code>LINES</code> only that were originally input as polygons. Any output <code>LINES</code> originally input as lines will not have this attribute when output.)
LEFT_EDGE_ATTR	The field into which is placed the ID of the line that forms the left edge in a winged-edge topology. Of the lines that connect to the arc's starting node, this is the first that is reached in a counter-clockwise direction from the arc. If neither this nor <code>RIGHT_EDGE_ATTR</code> is specified, a winged-edge topology is not computed.
RIGHT_EDGE_ATTR	The field into which is placed the ID of the line that forms the right edge in a winged-edge topology. Of the lines that connect to the arc's ending node, this is the first that is reached in a clockwise direction from the arc. If neither this nor <code>LEFT_EDGE_ATTR</code> is specified, a winged-edge topology is not computed.
NODES_ATTR	See <code>PROPAGATE_ATTRS</code> (<code>NODES_ATTR</code> is only applicable when this clause is specified).

Clause	Description
PROPAGATE_ATTRS	If this option is specified, attribute lists from the relevant input features for each output are created. For each node, this will be a list of lines and a list of polygons touching the node; for lines, there will be a list of nodes and a list of polygons; and for polygons, list of nodes and list of lines. The attribute names given by NODES_ATTR, ARCS_ATTR, and AREA_ATTR will be used as the base names for the lists.
POLYGONS_ATTR	The field into which all the IDs of the polygons which are either on the left or right of this line are placed. The left side is defined to be that polygon which is on the left of the line when travelling from the FROM_NODE to the TO_NODE along the arc. The right side is defined to be that polygon which is on the right of the line when travelling from the FROM_NODE to the TO_NODE along the arc. The UNIVERSE_POLYGON is given an ID of 0. All the IDs are listed in a comma separated character string, with the IDs of the polygons on the left having a negative sign in front of them. Typically, this list will have two entries, but in invalid coverages this list could hold more than two IDs. (This attribute will appear on output LINES only that were originally input as polygons. Any output LINES originally input as lines will not have this attribute when output.)
POLYGON_NUMBER_ATTR	The unique identifier assigned to each polygon which is output from the factory, numbered increasing from 1. The universal polygon is given an identifier of 0. (This attribute will appear on output POLYGONS and UNIVERSE_POLYGON only.)
ARCS_ATTR	Output for each polygon, this is the name of the attribute that contains the list of the IDs of the arcs which form the polygon's boundary. All the IDs are listed in a comma-separated character string. If a given arc has the opposite orientation to the respective segment in the polygon boundary, its ID will have a negative sign in front of it. If an output polygon has multiple pieces (that is, has a hole), then the arc numbers for the outer shell will be first, followed by the number "0" indicating the end of the arc list for the outer polygon. The list of arcs for the first hole would then be followed by another "0" and so on until all holes have been represented. (This attribute will appear on output POLYGONS and UNIVERSE_POLYGON only.)
PERIMETER_ATTR	The attribute into which the perimeter of the polygon will be stored. The UNIVERSE_POLYGON will have its perimeter given as negative. (This attribute will appear on output POLYGONS and UNIVERSE_POLYGON only.)

Clause	Description
AREA_ATTR	The attribute into which the area of the polygon will be stored. The UNIVERSE_POLYGON will have its area given as negative. (This attribute will appear on output POLYGONS and UNIVERSE_POLYGON only.)
NODE_NUMBER_ATTR	The fieldname that will be used to contain the number of the node, specified for node features. Numbered from 1 up. (This attribute will appear on output NODES only.)
ANGLE_PREFIX_ATTR	<p>The fieldname of the list attribute to contain information on all of the lines that intersect at this topologically significant location. The information that is placed on this list includes the angle the line segments intersect the node, the ID number of each line segment, and whether the segment is incoming to or outgoing from the node. (This attribute will appear on output NODES only.)</p> <p>Here is an example of what would appear on a node feature if attrInfo is specified as the value for ANGLE_PREFIX_ATTR.</p> <pre>attrInfo{0}.fme_node_id 14 attrInfo{0}.fme_node_angle 45 attrInfo{1}.fme_node_id -82 attrInfo{1}.fme_node_angle 90 attrInfo{2}.fme_node_id 61 attrInfo{2}.fme_node_angle180 attrInfo{3}.fme_node_id -112 attrInfo{3}.fme_node_angle304</pre> <p>The order the segment information appears in the list is the order in which they intersect the node, counterclockwise from the right horizontal axis. If the ID number is negative, this indicates the line's direction with respect to this node is incoming; a positive sign indicates the line is outgoing. In the example above, lines #82 and #112 are incoming, while lines #14 and #61 are outgoing.</p>
OVERLAPPING_DATA_PREFIX_ATTR	<p>If this keyword contains a non-empty value, it sets the factory into a mode where no collinear lines or overlapping points are output at all, whether they came from source linear features or from the borders of source area features or input points, or calculated as intersection points. In this mode, all output lines or points which were overlapping with at least one direct input will contain a list attribute with information about each input with which it was overlapping. This keyword sets the fieldname of the list attribute to contain all the attributes (that do not start with "fme_") from all of the input lines or points that were overlapping with the final output line or point.</p> <p>A side effect of this option is that only arcs that form part of a polygon boundary will be considered in the calculation of LEFT_EDGE_ATTR and RIGHT_EDGE_ATTR. (All arcs originating only from line input will have their own ID supplied as their left edge ID, and the negation of this as their right edge ID.)</p>

Clause	Description
START_EDGES_ONLY	If this keyword is specified, then a different feature is put out for each shell of the universe. As well, the "arcs" attribute on each polygon ring will only contain a single (unsigned) ID of one of the edges in the ring. For polygons, the "arcs" attribute will be a single element list with a single (unsigned) ID of one of the edges in that polygon. For output donuts, the "arcs" attribute list will contain the (unsigned) IDs, one from each ring, in order.
ALLOW_CYCLES	If this keyword is specified, then coordinate “cycles” within polygons are allowable and correct edge lists will be generated for such polygons. A “cycle” is an edge that occurs twice in the same polygon’s boundary—once forwards and once backwards. This edge’s ID will appear twice in the polygon’s edge list, positive in one instance and negative in the other.
ASSUME_CLEAN_DATA	If this keyword is specified and set to yes, then the input data is assumed to be correctly intersected and clean. In that case, no intersection is performed. Otherwise, the intersections of the data are computed prior to constructing the topology.

Assumptions

None.

Output Tags

The TopologyFactory supports the following output tags.

Tag	Description
LINE	All topologically significant lines are output. They have the attributes FROM_NODE_ATTR, TO_NODE_ATTR, RIGHT_POLY_ATTR, LEFT_POLY_ATTR and POLYGONS_ATTR specified as directed by the factory clauses.
POLYGON	The polygonal entities constructed are output here. These have a list of the LINES which make up the polygon along with the actual geometry.
NODE	All topologically significant nodes are output. They have the attributes NODE_NUMBER_ATTR and ANGLE_PREFIX_ATTR specified as directed by the factory clauses.

Example

The example below uses the `TopologyFactory` to construct the linear topology for the line features that are input. It also removes any duplicate arcs found within the features that are passed-in to it. The result is a set of linear features with all duplication removed and new attributes added to each line to indicate how it is connected to the other linework passed in to the `TopologyFactory`.

```

FACTORY_DEF * TopologyFactory                                \
  INPUT FEATURE_TYPE *                                       \
  ARC_NUMBER_ATTR arcNum                                     \
  FROM_NODE_ATTR fromNode                                   \
  TO_NODE_ATTR toNode                                       \
  RIGHT_POLY_ATTR rightPoly                                  \
  LEFT_POLY_ATTR leftPoly                                    \
  POLYGON_NUMBER_ATTR polyNum                               \
  ARCS_ATTR arc                                              \
  POLYGONS_ATTR poly                                         \
  OUTPUT LINE FEATURE_TYPE * @Log(line)                      \

```

TriangulationFactory

Syntax

```
FACTORY_DEF <ReaderKeyword> TriangulationFactory \
[FACTORY_NAME <factory name>] \
[INPUT FEATURE_TYPE <factory type> \
[<attribute name> <attribute value>]* \
[<feature function>]*]* \
[OUTPUT TRIANGLES FEATURE_TYPE <feature type> \
[<attribute name> <attribute value>]* \
[<feature function>]*]* \
[OUTPUT UNTOUCHED FEATURE_TYPE <feature type> \
[<attribute name> <attribute value>]* \
[<feature function>]*]*
```

Overview

Breaks an input geometry into triangular units. A Delaunay triangulation scheme is used. The triangulated features are output through the TRIANGLES port. Unprocessed geometries are output through the UNTOUCHED port.

For 2D geometries the triangulation is performed with respect to the X-Y plane.

For 3D geometries, such as faces, the triangulation is performed with respect to the normal direction of each surface.

Assumptions

None.

Output Tags

The TriangulationFactory supports the following output tag.

Tag	Description
TRIANGLES	Output features with triangulated geometries.
UNTOUCHED	Output features with geometries that were not processed by triangulation.

Geometry Type of Output Geometry Container

The geometry type of the output geometry container varies depending on the geometry type of the input geometry:

- If the input is an IFMETriangleStrip, IFMETriangleFan, IFMEFace or IFMEPolygon that is already triangular, the output will be a copy of the input geometry and hence has the same geometry type as the input.
- If the input is a non-triangular IFMEArea or IFMEMultiArea, the output will be an IFMEMultiArea of triangles.
- If the input is a non-triangular IFMEFace, or IFMERectangleFace, the output will be an IFMECompositeSurface, whose components are triangular IFMEFaces.
- If the input is an IFMECompositeSurface, the output will be an IFMECompositeSurface. The type of each component in the resulting IFMECompositeSurface is determined as described above.
- If the input is an IFMEMultiSurface, the output will be an IFMEMultiSurface. The type of each component in the resulting IFMEMultiSurface is determined as described above.
- If the input is an IFMEBox, IFMEExtrusion, or IFMEBRepSolid, the output will be an IFMEBRepSolid. The type of each component in the resulting IFMEMultiSurface is determined as described above.
- If the input is an IFMECSGSolid, the type of the output is determined by the constructed result described by the CSG solid hierarchy which could be an IFMEMultiSolid, IFMEBRepSolid, or IFMENull if the result is an empty set.
- If the input is an IFMECompositeSolid, the output will be an IFMECompositeSolid. The type of each component in the resulting IFMECompositeSolid is determined as described above.
- If the input is an IFMEMultiSolid, the output will be an IFMEMultiSolid. The type of each component in the resulting IFMEMultiSolid is determined as described above.
- If the input is an IFMEAggregate, the output will be an IFMEAggregate. The type of each component in the resulting IFMEAggregate is determined as described above.

Examples

This example shows the basic use of the factory.

```

FACTORY_DEF * TriangulationFactory      \
    FACTORY_NAME TriangleGenerator      \
    INPUT  FEATURE_TYPE Triangulation_INPUT      \
    OUTPUT * FEATURE_TYPE Triangulation_OUTPUT      \
    OUTPUT UNTOUCHED FEATURE_TYPE Untouched_OUTPUT

```


VectorOnRasterOverlayFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```

FACTORY_DEF <ReaderKeyword>
    VectorRasterOverlayFactory
    [FACTORY_NAME <factory name>]
    [INPUT FEATURE_TYPE <feature type>
        [<attribute name> <attribute value>]*
        [<feature function>]* ]*
    ACTION <processing type>
    [GROUP_BY [<field name>]+]*
    [OUTPUT VECTOR
        FEATURE_TYPE <feature type>
        [<attribute name> <attribute value>]*
        [<feature function>]*]*

```

Overview

This factory was designed to be used in situations where processing is done with a single raster and many vectors. The processing pattern is defined as follows: for each group, incoming valid vector features are stored until a reference raster is received for this group. Immediately following the reception of this raster, any stored vectors are processed using the raster. The vectors that come after reception of the raster are processed as they are received. Multiple rasters entering the factory in the same group will generate an error.

The processing that will take place depends on the value specified for the **ACTION** parameter. However, every scenario will involve outputting one vector feature for every vector feature that the factory receives. A single different raster may be used for every individual group of incoming features. The **GROUP_BY** parameter should be used for the purpose of defining such feature groups.

Currently, the factory supports only the following **ACTION**:

POINT_ON_RASTER

In this mode, the factory requires points as input vectors, and numeric rasters as input rasters. For each point input, a point is output. The data contained on each output point are the attributes from the input point and raster features and the z value from the raster at the location of the input point.

Assumptions

For the **POINT_ON_RASTER** processing type, incoming line features must be three-dimensional. The incoming raster feature must be a numeric raster

containing elevation data; color rasters are not valid as input to this factory, and must be converted to numeric rasters first.

Output Tags

Tag	Description
VECTOR	The vector created for each input vector feature will be output via the port associated with this tag. Note that the output features will contain any attributes that the corresponding input vector features had, as well as attributes contained on the reference input raster.

VectorToRasterFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> VectorToRasterFactory
[FACTORY_NAME <factory name>]
[INPUT (VECTOR|RASTER)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
[ANTI_ALIASING (yes|no)]
[TOLERANCE <float>]
[HEIGHT <number of rows>]
[WIDTH <number of columns>]
[X_CELL_SIZE <x cell spacing>]
[Y_CELL_SIZE <y cell spacing>]
[GROUND_EXTENTS_MINX <minX>]
[GROUND_EXTENTS_MINY <minY>]
[GROUND_EXTENTS_MAXX <maxX>]
[GROUND_EXTENTS_MAXY <maxY>]
[INTERPRETATION <interpretation>]
[BACKGROUND_COLOR <fme color>]
[BACKGROUND_VALUE <numeric value>]
[FILL_WITH_NODATA (yes|no)]
[GROUP_BY [<attribute name>+]*]
[OUTPUT (RASTER)
    FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory takes a set of point, linear, polygonal and/or donut features, and renders them onto the resulting raster. If a raster feature is supplied on the *RASTER* input tag, the properties of the output raster are taken from that input raster. If not, the properties of the resulting raster will be based on the various input options.

The size of the output raster can be determined by specifying either both *HEIGHT* and *WIDTH*, or both *X_CELL_SIZE* and *Y_CELL_SIZE*. These clauses are only applicable when no raster feature is supplied.

The *GROUND_EXTENTS_** clauses may optionally be used to specify the extents of the output raster. All four keywords must be specified in order for them to be applied. If the extents are not explicitly specified, the output raster extents will be determined by the union of the bounding boxes of the valid input vector features. This clause is only applicable when no raster feature is supplied.

The *INTERPRETATION* parameter sets the interpretation of the output raster. This clause is only applicable when no raster feature is supplied. Valid interpretations are *RGB24*, *RGB48*, *RGBA32*, *RGBA64*, *RED8*, *RED16*, *GREEN8*,

GREEN16, BLUE8, BLUE16, ALPHA8, ALPHA16, GRAY8, GRAY16, INT8, INT16, INT32, INT64, UINT8, UINT16, UINT32, UINT64, REAL32, and REAL64. If this clause is unspecified and no raster feature is provided, the first vector feature will be used to determine the type of the raster. If it has an `fme_color` attribute, then a raster with interpretation `RGB48` will be created. Otherwise, if it has `Z` coordinate values, a raster with interpretation `REAL64` will be created.

In order to be drawn on color bands, vector features must have an `fme_color` attribute. Pixel values for red, green, and blue bands will be taken from the corresponding component of a feature's `fme_color` attribute. Pixel values for gray bands will be the average of the `fme_color` components. Pixel values for alpha bands are not taken from the `fme_color` attribute and will always be the maximum value of the data type, to represent full opacity.

In order to be drawn on numeric bands, vector features must have have `Z` coordinates. Pixel values for numeric bands will be taken from the `Z` coordinates.

The `BACKGROUND_COLOR` and `BACKGROUND_VALUE` clauses control the fill value that will be used to fill portions of the raster containing no vector input. `BACKGROUND_COLOR` should be used for color interpretations and `BACKGROUND_VALUE` should be used for numeric interpretations. If `FILL_WITH_NODATA` is set to `yes`, the background fill color or fill value will also be flagged as the nodata value for each raster band. These clauses are only applicable when no raster feature is supplied.

Specifying the parameter `ANTI_ALIASING` as 'Yes' will perform line smoothing on the drawn lines. Setting the option to 'No' will not perform this smoothing. When anti-aliasing is not set, the `TOLERANCE` parameter must be given a value; this parameter is the maximum normalized distance from a line segment or polygon vertex to a pixel to be rendered. A `TOLERANCE` of 1.0 will turn on all pixels touched by a line, while a `TOLERANCE` of 0.0 will only turn on those pixels where the line passes directly through the center.

The input features may be partitioned into groups based on attribute values using the `GROUP_BY` clause and one raster feature is output for each group. If the `GROUP_BY` clause is not specified, then all input features will be processed together and a raster will be output. Attributes specified in the `GROUP_BY` clause are carried across from the `INPUT` features to the `OUTPUT RASTER` features.

This factory currently supports non-planar 3D lines but will render all 3D polygons as planar polygons.

Assumptions

None.

VirtualEarthTileFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> VirtualEarthTileFactory      \
  [FACTORY_NAME <factory name>]                            \
  [INPUT FEATURE_TYPE <factory type>                       \
    [<attribute name> <attribute value>]*                    \
    [<feature function>]*]**                                \
  [MIN_ZOOM_LEVEL <minimum zoom level>                     \
  [MAX_ZOOM_LEVEL <maximum zoom level>                     \
  [INTERPOLATION_TYPE_NAME [nearestneighbor|bilinear|      \
    bicubic|average4|average16]                             \
  [QUADKEY_ATTRIBUTE <attribute name>]                     \
  [RASTER_INDEX_ATTRIBUTE <attribute name>]                \
  [OUTPUT TILES FEATURE_TYPE <feature type>               \
    [<attribute name> <attribute value>]*                    \
    [<feature function>]*]**

```

Overview

This factory is used to create a series of image tiles that can be overlain in Virtual Earth. This is done by resampling rasters to various different resolutions and then splitting them into tiles. This factory produces similar output to the MSR MapCruncher utility.

The `MIN_ZOOM_LEVEL` and `MAX_ZOOM_LEVEL` clauses specify the zoom levels for which tiles will be generated. Level 1 is the lowest level of detail; at that level, the entire world map is 512 x 512 pixels. Each level increases the number of rows and columns by a factor of two: level 2 is 1024 x 1024, level 3 is 2048 x 2048, etc. Level 23 is the highest level of detail. If `MIN_ZOOM_LEVEL` is unspecified, the minimum zoom level will default to 1. Note that tiles will not be generated if the input raster covers less than 1 row and 1 column at a particular zoom level, regardless of the value of `MIN_ZOOM_LEVEL`. If `MAX_ZOOM_LEVEL` is unspecified, the maximum zoom level will be the smallest zoom level such that the resampled raster has more rows or columns than the original raster.

The `INTERPOLATION_TYPE_NAME` clause specifies the interpolation method to use when resampling to produce the different levels of detail. This clause is optional; it defaults to `NEARESTNEIGHBOR`.

The `QUADKEY_ATTRIBUTE` specifies the attribute in which the quadkey of each tile should be stored. Virtual Earth quadkeys uniquely identify a single tile at a particular level of detail. Generally, when writing out the rasters generated by this factory, one would fanout the destination feature type on this attribute. The PNG writer is recommended for the best results.

If the `RASTER_INDEX_ATTRIBUTE` clause is specified, an attribute will be added to each output tile that identifies which raster it was created from. This index is

zero-based, so all tiles created from the first input raster will have a value of 0, all tiles created from the second input raster will have a value of 1, etc.

This factory accepts only features that have raster geometry and is unaffected by raster band and/or palette subselection.

Additionally, this factory only accepts features that are in the EPSG:3785, EPSG:900913, or SPHERICAL_MERCATOR coordinate systems. All features must be reprojected into one of these coordinate systems prior to entering this factory.

Attributes will be carried across from the INPUT features to the respective TILES features.

Assumptions

None.

Output Tags

The `VirtualEarthTileFactory` supports the following output tag.

Tag	Description
TILES	Output Virtual Earth tiles created from the input feature(s).

Examples

In the following example, Virtual Earth tiles will be generated for zoom levels between 5 and 18. The quadkey for each output feature will be stored in the `_quadkey` attribute.

```

FACTORY_DEF * VirtualEarthTileFactory \
  FACTORY_NAME VirtualEarthTiler \
  INPUT FEATURE_TYPE my_raster \
  MIN_ZOOM_LEVEL 5 \
  MAX_ZOOM_LEVEL 18 \
  QUADKEY_ATTRIBUTE _quadkey \
  OUTPUT TILES FEATURE_TYPE VIRTUALEARTHTILER_TILES

```


WarpFactory

Note: This factory is not supported by FME Base Edition.

Syntax

```

FACTORY_DEF <ReaderKeyword> WarpFactory
  [FACTORY_NAME <factory name>]
  [INPUT (CONTROL_VECTOR|CONSTRAINT_LINE|OBSERVED)
  FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*
  WARP_METHOD (AFFINE|RUBBER_SHEET)
  [OBSERVED_FEATURESTORE <feature store file>]
  [CONTROLVECTOR_FEATURESTORE <feature store file>]
  [MAX_DISTANCE <distance>]
  [MAX_POINTS <max pts>]
  [EXPONENT_WEIGHT <float>]
  [OUTPUT (CONTROL_VECTOR|CORRECTED|BAD_CONTROL)
  FEATURE_TYPE <feature type>
    [<attribute name> <attribute value>]*
    [<feature function>]*]*

```

Overview

This factory performs warping operations on the spatial coordinates of features. It is used to adjust a set of observed features so they more closely match a set of reference features in space.

As input, it takes three sets of features: one representing the control features used to compute the corrections, one representing the observed features to be corrected, and an optional set of constraint lines, which define boundaries across which control vectors have no influence on observed features. Output consists of the warp-corrected features.

Each input control feature accepted by the `CONTROL_VECTOR` input clause represents a control vector whose start point is at an observed data vertex and whose end point is at the corresponding reference data vertex. The control vector represents the correction required to go from the observed vertex to the reference vertex. (Control vectors with only one point are interpreted as a requirement that this location not change from the observed dataset to the reference dataset. This is often referred to as a *tie point*.)

The factory currently supports two methods of warping correction. Both methods use the same control vectors in computing the correction.

The first method, `AFFINE`, computes and applies a 6-parameter affine transform to the observed input data to obtain the corrected output features. This computation requires that a minimum of four control vectors are input to the `WarpFactory`. See the `@Affine` function description in this manual for a basic

discussion of the affine transform itself. Affine correction can be used to apply combined linear translations, rotations, scalings, and reflections to the input data.

The second method of warping, `RUBBER_SHEET`, computes and applies a correction to each observed point using weighted values based on the distance between the point and the control vector start points. This has the effect of moving each observed point as if it were located on the surface of a rubber sheet.

`EXPONENT_WEIGHT` is a numeric weighting factor used by the `RUBBER_SHEET` warp method to determine the strength of the warping correction. The greater the value, the greater the degree of correction. Although `EXPONENT_WEIGHT` can be set to any floating point number, practical values typically range from 1.0 to 2.0. Experimentation can be used to determine the best value.

If the `EXPONENT_WEIGHT` clause is absent, the `RUBBER_SHEET` method uses a default value of 2.0. The `AFFINE` method ignores `EXPONENT_WEIGHT` if present.

`MAX_DISTANCE` indicates the influence of control vectors. Any control vector start point farther than `MAX_DISTANCE` from the point being operated on will have no effect in the correction computation. If `MAX_DISTANCE` is not specified, then all control vectors will be used for correcting every point. (`MAX_DISTANCE` is currently supported only for `RUBBER_SHEET` warping.)

`MAX_POINTS` indicates that only the closest `MAX_POINTS` vectors will have an effect on any point being warped. If not specified, then all control vectors will be used for correcting every point. (`MAX_POINTS` is currently supported only for `RUBBER_SHEET` warping.)

If both `MAX_DISTANCE` and `MAX_POINTS` are specified, then the more limiting clause will take precedence. That is, only the closest `MAX_POINTS` vectors which are within a distance of `MAX_DISTANCE` from a given point will have an influence on that point.

In addition to the above, the effects of control vectors on a rubber sheeting process may be further limited through use of `CONSTRAINT_LINES`. `CONSTRAINT_LINES` are boundaries, across which control vectors have no influence on points of observed features. If the “line of sight” from a point on an observed feature to the starting point of a control vector crosses a constraint line, that control vector will not affect the resulting warped position of the point in question. If the line of sight touches the end of a constraint line, or either the control vector or observed point is actually located on the constraint line, then the control vector will still influence the observed point.

Notes It is possible for RUBBER_SHEET warping to break apart polygons or to reposition points originally within polygons to locations outside of them after the warp. AFFINE warping preserves polygons and point-in-polygon relationships.

MAX_DISTANCE, MAX_POINTS and CONSTRAINT_LINES function only with the RUBBER_SHEET interpolation, not the AFFINE.

The WarpFactory must wait until all input is read before it can do any processing. This can slow down translations if the data volumes are large. To help alleviate this, the OBSERVED_FEATURESTORE and CONTROLVECTOR_FEATURESTORE clauses can be used to specify an FME feature store file (produced by the FFS writer or the RecordingFactory) that contains the data required to perform warping. If one of these clauses is specified, the corresponding features come from the specified recording; otherwise, the data enters the factory in the usual way.

Output Tags

The WarpFactory supports the following output tags.

Tag	Description
CONTROL_VECTOR	Valid control vector features (those which are 2-point lines) have this specification applied to them as they are output. A tie point is input as a point feature.
CORRECTED	Warp-corrected features have this specification applied to them as they are output.
BAD_CONTROL	Invalid control vector features (those which are not points or 2-point lines) have this specification applied to them as they are output.

Control Vector Creation

The following steps outline a suggested procedure for the creation of control vectors for a MapInfo format data set. Similar procedures can be used for other formats, such as ESRI Shape.

- 1 Open a map window displaying the observed and reference data sets as read-only, each in a different line color.
- 2 Create a new writable table for the control vectors and add it to the existing map window as another layer. Set this layer to show the direction for the lines. Add an attribute to this new table. It does not matter what attribute is added, since it is not required for the warping process. It is only needed to conform to MapInfo requirements that a table have at least one attribute.

- 3 Turn snapping on and add at least four 2-point lines to the control vector layer, each one going from an observed to a corresponding reference location. In general, the more control vectors, the better. They should be as evenly distributed across the coverage area as possible.
- 4 Once all control vectors have been entered, examine their directions. If they all seem to be pointing in the same direction, or go around in a circle, or expand outward or inward from a point, the `AFFINE` method may give a better correction than the `RUBBER_SHEET` method. If the directions seem to be essentially random, the `RUBBER_SHEET` method may be better.
- 5 Save the control vector table into the same directory that contains the observed data set.
- 6 In the `WarpFactory` code in the FME mapping file, specify this table as the one that supplies the `CONTROL_VECTOR` features.

Example

The following example illustrates the use of the `WarpFactory`. It accepts features in MapInfo TAB format. Control vectors are supplied from a TAB layer called `controlVectors` and the observed features to be corrected are supplied from a layer called `originalStreets`. The `RUBBER_SHEET` method is specified with an `EXPONENT_WEIGHT` value of 1.8 and corrected features are output as layer `goodStreets`.

```

FACTORY_DEF MAPINFO WarpFactory                                \
  INPUT CONTROL_VECTOR FEATURE_TYPE controlVectors            \
  INPUT OBSERVED FEATURE_TYPE originalStreets                  \
  WARP_METHOD RUBBER_SHEET                                     \
  EXPONENT_WEIGHT 1.8                                          \
  OUTPUT CORRECTED FEATURE_TYPE goodStreets

```


XFMapFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> XFMapFactory \
  [FACTORY_NAME <factory name>] \
  [INPUT FEATURE_TYPE <feature type> \
    [<attribute name> <attribute value>]* \
    [<feature function>]*]+ \
  [XML_FILE <attribute name>| XML_STREAM <attribute name>] \
  [XFMAP_FILE <attribute name or value>]+ \
  [XFMAP_STREAM <attribute name or value>]+ \
  [XRS <value>] \
  [XML_SOURCE_NAME <value>] \
  [XML_SOURCE_VALUE <value>] \
  [FEATURE_COUNT_PER_XML <value>] \
  [FEATURE_TYPE_ATTR <value>] \
OUTPUT (MAPPED|INVALID) FEATURE_TYPE <feature type>\
  [<attribute name> <attribute value>]* \
  [<feature function>]*]*

```

Overview

This factory constructs features from XML documents. The XML data to map may be specified wholly as a string with the **XFMAP_STREAM** clause or as a file path with the **XML_FILE** clause.

The XFMapFactory uses a set of rules to map XML data into FME features. These mapping rules are defined in xfMap documents. One or more xfMaps can be specified for this transformer via the **XFMAP_FILE** and/or **XFMAP_STREAM** clauses. See the XML reader documentation for more information regarding the xfMap.

An XRS (XML Reader Switch) document allows the XFMapFactory to automatically configure itself to read “known” XML datasets without the need to specify in advance the appropriate xfMap(s). If no xfMap and no XRS are specified for this factory, then the default XRS document is used. The default XRS document is named xrs.xml, and it is located in the xml/xrs subdirectory of the FME installation directory. See the XML reader documentation for more information regarding the XRS.

The optional **XML_SOURCE_NAME** and **XML_SOURCE_VALUE** clauses specify the name and the value for the attribute that is to be added to the mapped features to identify the XML source document. The default value for this attribute depends on the clause that was used to specify the XML document, the default value is the file path for the **XML_FILE** clause or the empty string for the **XML_STREAM** clause.

The optional **FEATURE_COUNT_PER_XML** clause sets the name for the attribute that enumerates features mapped per XML document.

The optional FEATURE_TYPE_ATTR clause sets the attribute to store the feature type for the mapped features.

Assumptions

None

Output Tags

The XFMap factory supports the following output tags.

Tag	Description
MAPPED	The features extracted from the XML source documents are output through the MAPPED output tag.
INVALID	The INPUT features containing an error in their XML source are output through the INVALID output tag.

Examples

```
MACRO XML_SOURCE "C:\xml\data.xml "  
MACRO XFMAP "C:\xml\xfmap.xml "  
  
FACTORY_DEF * CreationFactory \  
    CREATE_AT_END no \  
    OUTPUT FEATURE_TYPE my_feature  
  
FACTORY_DEF * SamplingFactory \  
    INPUT FEATURE_TYPE my_feature \  
        @SupplyAttributes(xml_source,$(XML_SOURCE)) \  
        @SupplyAttributes(xfmap,$(XFMAP)) \  
    SAMPLE_RATE 1  
  
FACTORY_DEF * XFMapFactory \  
    INPUT FEATURE_TYPE my_feature \  
    XML_FILE xml_source \  
    XFMAP_FILE xfmap \  
    FEATURE_COUNT_PER_XML _feature_count_ \  
    FEATURE_TYPE_ATTR _feature_type_ \  
    OUTPUT MAPPED FEATURE_TYPE mapped_feature \  
    OUTPUT INVALID FEATURE_TYPE invalid_feature
```

XPathFactory

Syntax

```

FACTORY_DEF <ReaderKeyword> XPathFactory \
  [FACTORY_NAME <factory name>] \
  [INPUT FEATURE_TYPE <feature type> \
    [<attribute name> <attribute value>]* \
    [<feature function>]*]* \
  MODE (EXTRACT|EXPLODE) \
  ( XPATH <XPath expression>[<XPath expression>]* \
    | XPATH_ATTRIBUTE <attribute name>[<attribute name>]*) \
  XML_ATTRIBUTE <attribute name> \
  RESULT_ATTRIBUTE <attribute name>[<attribute name>]* \
  [WRITE_XML_HEADER (yes|no)] \
  [EXTRACT_AS (LIST_ATTR|SINGLE_ATTR)] \
  [SEPARATOR_VALUE (separator)] \
  [EXPLODED_XPATH <attribute name>] \
  OUTPUT (EVALUATED|INVALID) \
    FEATURE_TYPE <feature type> \
    [<attribute name> <attribute value>]* \
    [<feature function>]*]* \

```

Overview

This factory uses XPath Expressions to extract values from XML stored in a feature attribute (EXTRACT mode), or to convert the XML into new features (EXPLODED mode). The XPATH / XPATH_ATTRIBUTE clause specifies the XPath expression or a feature attribute containing an XPath expression. Only one of XPATH and XPATH_ATTRIBUTE can be set at a time. The XML_ATTRIBUTE clause specifies the feature attribute that contains the XML, and the RESULT_ATTRIBUTE clause specifies the feature attribute that will store the evaluated XPath result, which could be anything from attribute values to XML document fragments. Users may choose to write out a XML header on the returned XML document fragments or not by setting WRITE_XML_HEADER. If this clause is not set, XPathFactory will not write any XML headers.

Multiple XPath expressions can be evaluated by the same XPathFactory if the XPATH / XPATH_ATTRIBUTE clause is specified with quoted and space separated XPath expressions or attribute names. The RESULT_ATTRIBUTE clause must be set so that the number of attribute names matches the number of XPath expressions/attributes specified. Each result attribute will correspond to an XPath expression/attribute by the order they are specified.

Optional clauses EXTRACT_AS and SEPARATOR_VALUE are only available in EXTRACT mode and EXPLODED_XPATH is only available in EXPLODE mode.

XPath Expressions

The XPath language is based on the XML tree representation. It provides the ability to navigate through the document and select nodes by a variety of criteria. This factory supports XPath1.0 as outlined in W3C Recommendation <http://www.w3.org/TR/xpath>.

By using different functions and operators of XPath, users could get results ranging from XML document fragments to attribute values. The following examples illustrates some basic use of XPath expressions.

book.xml (Sample XML input document)

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<bookstore>

<book category="COOKING">
  <title lang="en">Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
</book>

<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>

<book category="WEB">
  <title lang="en">XQuery Kick Start</title>
  <author>James McGovern</author>
  <author>Per Bothner</author>
  <author>Kurt Cagle</author>
  <author>James Linn</author>
  <author>Vaidyanathan Nagarajan</author>
  <year>2003</year>
  <price>49.99</price>
</book>

<book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
</book>

</bookstore>
```

XPath Expression: /bookstore/book/title/

```

<title lang="en">Everyday Italian</title>
<title lang="en">Harry Potter</title>
<title lang="en">XQuery Kick start</title>
<title lang="en">Learning XML</title>

```

XPath Expression: /bookstore/book/title/text()

```

Everyday Italian
Harry Potter
XQuery Kick start
Learning XML

```

XPath Expression: /bookstore/book[@category='WEB']

```

<book category="WEB">
  <title lang="en">XQuery Kick Start</title>
  <author>James McGovern</author>
  <author>Per Bothner</author>
  <author>Kurt Cagle</author>
  <author>James Linn</author>
  <author>Vaidyanathan Nagarajan</author>
  <year>2003</year>
  <price>49.99</price>
</book>

<book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
</book>

```

XPath Expression: /bookstore/book/@category

```

COOKING
CHILDREN
WEB
WEB

```

Extract Mode

When the MODE clause is set to EXTRACT, the factory will evaluate the XPath expression and add the result as an attribute specified by RESULT_ATTRIBUTE. If there are more than one XPath and result attribute, the factory treats each XPath and result attribute as a separate pair.

An option in this mode allows the users to decide whether the returned result from an XPath should be merged together and loaded into a single attribute or should be loaded into individual attributes for each node returned. The EXTRACT_AS clause serves to specify whether SINGLE_ATTR or LIST_ATTR should be added. If SINGLE_ATTR is selected, the result will be

added on to the result attribute specified by the corresponding `RESULT_ATTRIBUTE` parameter. This result will contain all nodes returned. In order to interpret the results better, the user may choose to add a separator (e.g. a comma) between each node in the result by specifying `SEPARATOR_VALUE`. Separators are only available when `SINGLE_ATTR` is set. On the other hand, if `LIST_ATTR` is set, the result returned will be added onto the feature as a list attribute with the name specified in `RESULT_ATTRIBUTE`. For each node returned, a member of the list attribute will be added. If `EXTRACT_AS` is not set, XPathFactory will extract as if `SINGLE_ATTR` is set with no separators.

Explode Mode

When the `MODE` clause is set to `EXPLODE`, the factory will create new features for each node returned and load the XML text of that node into an attribute specified by the corresponding `RESULT_ATTRIBUTE` argument. In order to identify the resulting features, users may choose to tag these new features by adding a new tag attribute with the XPath expression used as the value. `EXPLODED_XPATH` specifies the new attribute name that users want all resulted features to be tagged with.

Output Tags

The `XPathFactory` supports the following output tags.

Tag	Description
EVALUATED	If a feature has been successfully evaluated in either <code>EXTRACT</code> or <code>EXPLODE</code> mode, they will be output through this tag.
INVALID	If a feature failed at any stage of processing (e.g. invalid XPath expression), it will be output through this tag.

Examples

The examples below illustrate the usage of the `XPathFactory` in `EXTRACT` and `EXPLODE` modes.

The first example uses the factory in `EXTRACT` mode. It evaluates two `XPATH` expressions into two `RESULT_ATTRIBUTES`. The results from `XPATH` expression `"/bookstore/book/title"` will be loaded into the list attribute `titles`. Hence, the feature will have `n` attributes added to it, i.e. `titles{0}`, `titles{1}`...`titles{n}`. Each `titles{}` attribute will have the serialized text of a result node. Similarly, the results from `XPATH` expression `"/bookstore/book/author/text()"` will be loaded into list

attribute `authors`, and added to the same input feature as `titles`. `WRITE_XML_HEADER` is set to 'yes' so all element node results will be added with a XML header. This means all the text in `authors` won't have a header because these are selected as text nodes. Finally, the modified feature will be output through the `EXTRACTED` output tag.

```

FACTORY_DEF * XPathFactory \
  INPUT FEATURE_TYPE * \
  MODE EXTRACT \
  XPATH "/bookstore/book/title" \
    "/bookstore/book/author/text()" \
  XML_ATTRIBUTE input \
  RESULT_ATTRIBUTE title author \
  WRITE_XML_HEADER yes \
  EXTRACT_AS LIST_ATTR \
  OUTPUT EVALUATED FEATURE_TYPE evaluated \
  OUTPUT INVALID FEATURE_TYPE *

```

The second example illustrates `EXPLODE` used in conjunction with `EXTRACT` by using two `XPathFactories`.

The first `XPathFactory` creates new features based on the two `XPaths` stored in feature attributes `xpath_title` and `xpath_author`. Some new features will be created with the result attribute `book_title` containing text result returned by evaluating `xpath_title`. These features will have another attribute named 'xpath_used' with the value of the `XPath` expression stored in 'xpath_title', because `EXPLODED_XPATH` is set. Similarly some new features will be created with only two attributes, `name_author` and `type`. These features will have also have an attribute named 'xpath_used', but with the value of the `XPath` expression stored in 'xpath_author'.

The second factory reads the 'title' attribute of the output from the first factory, evaluates with `XPath "/title/@lang"`. This will add a new attribute 'language' to those features succeed in evaluation. If XML text in the 'title' attribute of a feature has multiple '@lang' values, then they will be concatenate together with a comma in between.

```

FACTORY_DEF * XPathFactory \
  INPUT FEATURE_TYPE * \
  MODE EXPLODE \
  XPATH_ATTR xpath_title xpath_author \
  XML_ATTRIBUTE input \
  RESULT_ATTRIBUTE book_title name_author \
  WRITE_XML_HEADER yes \
  EXPLODED_XPATH xpath_used \
  OUTPUT EVALUATED FEATURE_TYPE exploded \
  OUTPUT INVALID FEATURE_TYPE *

FACTORY_DEF * XPathFactory \
  INPUT FEATURE_TYPE exploded \
  MODE EXTRACT \

```

```

XPATH "/title/@lang"
XML_ATTRIBUTE title
RESULT_ATTRIBUTE language
WRITE_XML_HEADER no
EXTRACT_AS SINGLE_ATTR
SEPARATOR_VALUE ", "
OUTPUT EVALUATED FEATURE_TYPE *
OUTPUT INVALID FEATURE_TYPE *

```

XQueryFactory

Syntax

```

FACTORY_DEF * XQueryFactory
[ FACTORY_NAME <factory name> ]
[INPUT SOURCE_DOCUMENT FEATURE_TYPE <feature type>
  [ <attribute name> <attribute value> ]*
  [ <feature function>]*]+
[INPUT QUERY_DOCUMENT FEATURE_TYPE <feature type>
  [ <attribute name> <attribute value> ]*
  [ <feature function>]*]+
[MODE (SINGLE_FEATURE|DYNAMIC_XML|DYNAMIC_QUERIES) ]
[XML_FILE <value> | XML_ATTR <value>]
[XQUERY_FILE <value> ]*
[XQUERY_EXPR <value> ]*
[XQUERY_ATTR <value> ]*
[EXTRACT_MODE yes]
[EXPLODE_MODE yes]
[EXTRACT_TYPE CSV|STRING ]
[SIMPLIFY_ON_SINGLE (yes|no) ]
[RESULTS_SEPARATOR <value>]
[QUERY_SEPARATOR <value>]
[TAG_QUERY_ATTRIBUTE <attribute name>]
[WRITE_HEADER yes|no] /
[COPY_INPUT_ATTRS (yes|no)]
[USE_W3C_UPDATE_EXTENSIONS [yes|no]]
[STATIC_QUERY (yes|no)]
[RESULT_ATTR <attribute name> ]
[OUTPUT (QUERY_RESULTS|INVALID_QUERY)
  FEATURE_TYPE <FEATURE_TYPE>
  [ <attribute name> <attribute value> ]*
  [ <feature function>]*]+ ]

```

Overview

This factory uses XQuery Expressions/Documents to extract values from XML documents stored in a file (XML_FILE) or as an attribute on a feature (XML_ATTR). Since XQuery expressions can contain references directly to the filesystem (as in ‘doc(“info.xml”)/node/text()’), neither XML_FILE or XML_ATTR need be set. At least one XQUERY_* must be specified.

Users may take a result set from an XQuery and join it together in a couple of ways (using EXTRACT_MODE and EXTRACT_TYPE), or may return separate features for every element in the result-set (using EXTRACT_MODE).

The XQueryFactory supports the XQuery Update Facility 1.0 Candidate Recommendation. If the XQUERY_* clause names an XQuery expression that should be parsed and evaluated as an update the clause USE_W3C_UPDATE_EXTENSIONS must be set to ‘yes’. An update expression either returns results, or performs an update but not both. For this

reason, it is not recommended to specify the update capabilities for the factory unless the query is performing an update as the resulting output may be inferior. For example, the elements of a result-set that is returned will be concatenated together. When the update capabilities are not being used more options are available for handling the result-set.

Clauses

Clauses	Description	Optional
XML_FILE <file path>	When this value is set the file is assigned as the target of the query. For a regular XQuery, the result set will be returned as a (set of) values. If performing an XQuery Update, then there is no result, and the query will result in an update directly to the file. Default: None Example: XML_FILE "info.xml"	yes
XML_ATTR <attribute name>	Specifies the input attribute name. The contents of the attribute on a feature should be a valid XML document. Default: None Example: XML_ATTR xml_string	yes
XQUERY_FILE <file value>	Specifies a file whose content is a valid XQuery document Default: None Example: XQUERY_FILE query.xqr	yes
XQUERY_ATTR <attribute name>	Specifies the name of an attribute whose content on the current feature is a valid XQuery document/expression. Default: None Example: XQUERY_ATTR xquery_string	yes
XQUERY_EXPR <value>	Specifies an XQuery expression/document directly. This clause is useful when the user wishes to apply a simple XQuery expression (possibly just an XPATH 2.0 expression) to every feature that goes through the factory. A limitation of this clause (over the XQUERY_FILE or XQUERY_ATTR clauses) is that the value of the XQUERY_EXPR clause must be in the system encoding. Default: None Example: XQUERY_EXPR '//geom/point'	yes

Clauses	Description	Optional
MODE (SINGLE_FEATURE DYNAMIC_XML DYNAMIC_QUERIES)	Specifies the way XML and XQUERY documents will be supplied to the factory. In SINGLE_FEATURE both documents are updated (since all information is available from the single feature). With DYNAMIC_XML, only the XML document is updated. With DYNAMIC_QUERIES the XML document is not updated, but the XQuery documents are. Default: SINGLE_FEATURE Example: MODE SINGLE_FEATURE	yes
RESULT_ATTR <attribute name>	Specifies the output result attribute name. The exact form of the attribute may differ depending on the EXTRACT_TYPE option Default: None	yes
USE_W3C_UPDATE_EXTENSIONS (yes no)	Specifies whether or not the queries are to be parsed and evaluated with the W3Cs XQuery Update Facility. Note: This should only be set to 'yes' if you are in fact processing an update. Default: no	yes
COPY_INPUT_ATTRS (yes no)	Specifies whether or not the input feature's attributes should be copied to any exploded output features. Only used if EXPLODE_MODE is set to 'yes' Default: no	yes
QUERY_SEPARATOR <value>	Specifies the string used to separate the result sets of separate queries for one another Default: ' ' Example; QUERY_SEPARATOR ##	yes
RESULTS_SEPARATOR <value>	Specifies the string used to separate the result sets of a query. Default: ',' Example; RESULTS_SEPARATOR	yes
EXTRACT_MODE (yes no)	When this clause is set to 'yes', the results of the queries are written onto the input feature, and it is sent out of the QUERY_RESULTS port Default: None	yes

Clauses	Description	Optional
EXTRACT_TYPE (STRING CSV)	<p>This clause specifies the shape of the results strings when the EXTRACT_MODE is set to 'yes'.</p> <p>CSV: Results are joined together into a CSV using the RESULTS_SEPARATOR, and these CSVs are assembled into a complex CSV using the QUERY_SEPARATOR.</p> <p>STRING: The results are simply concatenated together into a single string. This is useful if the query is intended to construct a valid XML document.</p> <p>Default: CSV</p>	yes
EXPLODE_MODE (yes no)	<p>When this clause is set to 'yes', the results of the queries are written into separate features for every result returned. These features are output from the factory via the QUERY_RESULTS port</p> <p>Default: None</p>	yes
WRITE_HEADER (yes no)	<p>When this clause is set to 'yes', the XML header is written onto the output results. Note that this does not effect XQuery Updates on files.</p> <p>Default: yes</p>	yes
SIMPLIFY_ON_SINGLE (yes no)	<p>When this clause is set to 'yes', and there is only a single extracted value, and the EXTRACT_TYPE is set to CSV then treat the EXTRACT_TYPE as STRING if there is only a single result for all queries.</p> <p>Default: no</p>	yes
STATIC_QUERY (yes no)	<p>When this clause is set to 'yes', if no input features are read then the factory will assume that the XQuery and XML documents are defined statically (either as files or literal expressions).</p> <p>Default: no</p>	yes

Extract Mode

When the EXTRACT_MODE clause is set to 'yes', the factory will evaluate the XQuery expression and place the result as an attribute specified by RESULT_ATTR. If there are multiple results, the results are concatenated together in the manner specified by the EXTRACT_TYPE parameter.

If the `EXTRACT_TYPE` parameter is set to ‘STRING’, the results are simply concatenated together. This is useful when you are, for example, constructing a new XML document from an existing one.

If the `EXTRACT_TYPE` parameter is set to CSV, the `RESULTS_SEPARATOR` parameter is used to separate the results, with the default value for this parameter being the comma character (‘,’). In the case that multiple XQuery expressions are defined, the results from individual queries are concatenated together using the `QUERY_SEPARATOR` parameter as the separator. The default value for the `QUERY_SEPARATOR` is the pipe character (‘|’).

Explode Mode

When the `EXPLODE_MODE` clause is set to ‘yes’, the factory will create new features for each element in the result-set of the XQuery expression. For example, an XQuery expression might just be an XPath 2.0 expression, in which case each node in the result set would be serialized and the resulting XML placed in the `RESULT_ATTR` attribute.

Output Tags

The `XQueryFactory` supports the following output tags.

Tag	Description
<code>QUERY_RESULTS</code>	If an XQuery expression has been evaluated successfully, any results of this expression will be output through this tag.
<code>INVALID</code>	If a feature failed at any stage of processing (e.g. invalid XQuery expression, malformed XML), it will be output through this tag.

Examples

Suppose that you have an XML document with the following structure:

EXAMPLE DOCUMENT 1:

```
<parcels>
  <parcel>
    <parcelLocation>
      <parcelBounds>
        <topLeft> 49.37208485655605 -123.18372130393982
      </topLeft>
        <bottomRight>49.370890232974375 -
123.18200469017029</bottomRight>
```

```

        </parcelBounds>
        <parcelRef>49.372716388370876 -123.1790542602539</
parcelRef>
        </parcelLocation>
        <parcelOwner>
        <owner>
        <name>Bob Smith</name>
        <address type="mail">2062 West 38 Ave, Vancouver,
BC, CANADA</address>
        </owner>
        </parcelOwner>
    </parcel>
    <parcel>
        <parcelLocation>
        <parcelBounds>
        <topLeft>49.37238665158286 -123.17986965179443 </
topLeft>
        <bottomRight> 49.37064012679701 -
123.17738056182861</bottomRight>
        </parcelBounds>
        <parcelRef>49.3715203830451 -123.17888259887695 </
parcelRef>
        </parcelLocation>
        <parcelOwner>
        <owner>
        <name>Alice Wight</name>
        <address type="mail">1037 West 36 Ave, Vancouver,
BC, CANADA</address>
        <address type="email">alice.wight@example.com</
address>
        </owner>
        </parcelOwner>
    </parcel>
</parcels>

```

Essentially, this is a simple format for (rectangular) parcels that includes a bounding box and a reference point, and some information about the owner. It is necessarily simple for expository purposes.

Here is an XQuery FLOWR expression that would construct a comma separated value string for the mailing addresses of owners that have email addresses.

```

for $owner in //owner
where $owner/address/@type = 'email'
return concat( $owner/name/text(),
               "|",
               $owner/address[@type="mail"]/text() )

```

A full factory that just extracted owner names might look like this

```

FACTORY_DEF * XQueryFactory \

```

```

FACTORY_NAME XQueryExploder \
INPUT SOURCE_DOCUMENT FEATURE_TYPE * \
XQUERY_EXPR //owner/name/text() \
XML_ATTR xml_doc \
MODE SINGLE_FEATURE \
EXTRACT_MODE yes \
EXTRACT_TYPE CSV \
RESULT_ATTR _result \
WRITE_HEADER No \
RESULTS_SEPARATOR ,
TAG_QUERY_ATTRIBUTE _query \
USE_W3C_UPDATE_EXTENSIONS no \
COPY_INPUT_ATTRS yes
OUTPUT QUERY_RESULTS FEATURE_TYPE \
    XQueryExploder_QUERY_RESULTS \
OUTPUT INVALID FEATURE_TYPE XQueryExploder_INVALID

```

If you wanted to construct a single feature for each result found, you would replace the parameter ‘EXTRACT_MODE yes’ with ‘EXPLODE_MODE yes’. Then each result name found would be written into the attribute ‘_result’ on a new feature. This would be useful if, for instance, you had a database of other information on people and you wanted to match it with the names of owners of parcels of land.

The XQueryFactory also supports the XQuery Update Facility 1.0 specified by the W3C, although this is currently only a Candidate Recommendation. Here is an example XQuery Update that replaces the contents of the example documents bounding box with a GML envelope (assuming that the XML_FILE was set to ‘example.xml’).

```

declare namespace gml = "http://www.opengis.net/gml" ;

for $node in //parcelBounds
return (
    delete node $node,
    insert node
        <gml:envelope>
            <gml:pos>{$node/topLeft/text()}</gml:pos>
            <gml:pos>{$node/bottomRight/text()}</gml:pos>
        </gml:envelope>
    as first into $node/..
)

```

Here is simple, but complete factory that inserts a last-modified date into the example document.

```

FACTORY_NAME XQueryExploder \
INPUT SOURCE_DOCUMENT FEATURE_TYPE * \
XQUERY_EXPR insert node <lastModified>20080921</
lastModified> as first into /* \
XML_ATTR xml_doc \
MODE SINGLE_FEATURE \

```

```
USE_W3C_UPDATE_EXTENSIONS yes \  
OUTPUT QUERY_RESULTS FEATURE_TYPE \  
  XQueryExploder_QUERY_RESULTS \  
OUTPUT INVALID FEATURE_TYPE XQueryExploder_INVALID
```

This simply inserts the tag '`<lastModified>20080921</lastModified>`' as the first child of the root element of current document.

XSLTFactory

Syntax

```

FACTORY_DEF <Reader Keyword> XSLTFactory \
[FACTORY_NAME <factory name>] \
[INPUT FEATURE_TYPE <factory type> \
[<attribute name> <attribute value>]* \
[<feature function>]*]* \
[FACTORY_STYLESHEET <file path>] \
(XML_DOC | XML_FILE | XML_STREAM) <attribute name> \
[(STYLESHEET_DOC | STYLESHEET_FILE | \
STYLESHEET_ATTRIBUTE) <attribute name>] \
STYLESHEET_PRECEDENCE (FACTORY_SS/FEATURE_SS) \
[(RESULT_FILE <attribute name>] \
[(RESULT_STREAM <attribute name>] \
[OUTPUT (TRANSFORMED/SKIPPED) \
FEATURE_TYPE <feature type> \
[<attribute name> <attribute value>]* \
[<feature function>]*]*

```

Overview

XSLT stands for XSL (eXtensible Stylesheet Language) transformations. This factory is used to transform an XML document into another XML document using an XSL stylesheet.

Input, output and stylesheet sources may be either a file path name or a string stored in an attribute. (Note that the factory asks for the attribute name that stores the file path instead of the file path itself.) The user can specify whether the input attribute is a file path or a string, by using **XML_FILE**, **XML_STREAM**, **STYLESHEET_FILE** and **STYLESHEET_STREAM** to explicitly declare **FILE** or **STREAM** types. Or if the attribute type is unknown, then the user may choose to use **XML_DOC** and **STYLESHEET_DOC** to let the factory decide whether it is a file or a string.

FACTORY_STYLESHEET sets a stylesheet that is common to all features passing into the factory. It will only be compiled once, hence provides faster transformations when applied to multiple documents.

STYLESHEET_ATTRIBUTE contains the attribute name that sets the feature stylesheet. **STYLESHEET_PRECEDENCE** determines if factory or feature stylesheet should be used. If **FEATURE_SS** is selected, but no feature stylesheet is available, then factory stylesheet will be used if there is one defined.

Each feature with an attribute of input XML document with a valid stylesheet will either have been transformed successfully (send to **TRANSFORMED** output) or unsuccessfully due to errors (send to **SKIPPED**). Transformed features will either output as a file with the file path stored in the attribute

specified in the RESULT_FILE clause, or have a new attribute added to it with the attribute name specified in the RESULT_STREAM clause (or both).

Assumptions

None

Clauses

Clauses	Description	Optional
FACTORY_STYLESHEET <file path>	Sets the factory stylesheet. This will be applied to all incoming features when FEATURE_PRECEDENCE is set to FACTORY_SS. Otherwise, it will apply to features that do not have a stylesheet available. Default: None Example: FACTORY_STYLESHEET "stylesheet.xml"	Yes
(XML_DOC XML_FILE XML_STREAM) <attribute name>	Specifies the input attribute name. The attribute can stores a file path name or an XML string. If XML_FILE is used, it will be treated as a file path. If XML_STREAM is used, it will be treated as a XML string stored in the attribute. However, if XML_DOC is used, the factory will try to determine whether it is a file or a stream for the user. Default: None Example: XML_DOC xml_string	No
(STYLESHEET_DOC STYLESHEET_FILE STYLESHEET_STREAM) <attribute name>	Specifies the feature stylesheet attribute name. The attribute can stores a file path name or an XSL string. If STYLESHEET_FILE is used, it will be treated as a file path. If STYLESHEET_STREAM is used, it will be treated as a XSL string stored in the attribute. However, if STYLEHSEET_DOC is used, the factory will try to determine whether it is a file or a stream for the user. Default: None Example: STYLESHEET_FILE table_style	Yes
STYLESHEET_PRECEDENCE (FACTORY_SS FEATURE_SS)	Specifies which stylesheet to use. Default: FACTORY_SS Example: STYLESHEET_PRECEDENCE FEATURE_SS	No

Clauses	Description	Optional
RESULT_FILE <attribute name>	Specifies the output result attribute name(s). If the attribute stores a file path, then use	Yes, but one of
&	RESULT_FILE. If the attribute will store the output xml string, then use RESULT_STREAM.	RESULT
RESULT_STREAM <attribute name>	Unlike input and stylesheet sources, RESULT_FILE and RESULT_STREAM can be set concurrently, resulting in both an output file and a new attribute on the processed feature.	keywords has to be set.
	Default: None	

Output Tags

The XSLTFactory supports the following output tags

Tag	Description
TRANSFORMED	Features containing XML file or XML String attribute transformed successfully using a XSLT stylesheet
SKIPPED	Features containing XML file or XML String attribute failed to transform due to errors, such as incorrect file paths.

Examples

In the following example, feature type “myxmlstring” is transformed from an input xml string type (stored in attribute “xml_string”) to a file (path stored in attribute “xslt_output”) and a xml string (stored in attribute “xslt_outstream”). “xsl_stylesheet” is the attribute that specifies feature stylesheet, which can either be a file or a xsl string in this case. The factory will attempt to decide whether the attribute contains a file path or string. If feature has a stylesheet available and stored in attribute “xsl_stylesheet”, then this stylesheet will be used because STYLESHEET_PRECEDENCE is set to FEATURE_SS. If no feature stylesheet is available, the factory stylesheet "C:\stylesheets\book_style.xsl" will be used.

```
FACTORY_DEF * XSLTFactory \
INPUT FEATURE_TYPE myxmlstring \
FACTORY_STYLESHEET "C:\stylesheets\book_style.xsl" \
XML_STREAM xml_string \
STYLESHEET_DOC xsl_stylesheet \
STYLESHEET_PRECEDENCE FEATURE_SS \
RESULT_FILE xslt_output \
RESULT_STREAM xslt_outstream \
OUTPUT TRANSFORMED FEATURE_TYPE *
```

/ /

About Workbench Transformers

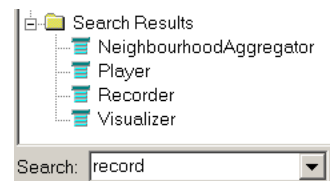
Within the FME Workbench environment, no distinction is made between FME functions and factories. Instead, all FME processing facilities are packaged into *transformers*, which operate on data and transform it in one way or another. Just like functions and factories, some transformers add attributes to features, others erase attributes, and still others operate on the geometry. Transformers may operate on individual features one at a time, or on groups of features.

To assist you in locating transformers and becoming familiar with their functionality, the Workbench on-line help files describe each transformer in greater detail. This chapter also provides cross-references between Workbench transformers and FME functions and factories. Experienced FME mapping file creators can use these cross-references to find the transformer that corresponds to a particular FME function or factory. Workbench users can use the cross-reference to find out which FME function or factory is used to implement a transformer, and then examine the documentation for that function or factory for even more detailed information on its operation.

Finding the right transformer for the task at hand can still be challenging. The Transformer gallery in FME Workbench provides a convenient keyword search facility for locating transformers – enter a keyword, transformer name, FME function, or FME factory to display a list of matches that can be easily placed in a workspace.

Tip

Search for a keyword in the Transformer gallery. The results will show all Transformers that match your keyword.



Transformer Help

For detailed information on transformers, open the Workbench Help menu and find *Transformers Help*. This folder contains separate detailed explanations for each transformer.

Transformer Cross-References

In most cases, FME functions and factories corresponds directly with a Workbench transformer.

Both the Workbench Help files and the Workbench Transformer Gallery list corresponding functions and factories in each transformer help topic. You can search the Gallery by function, factory, transformer, or keyword.

Quick Reference

Functions

Geometry and Coordinates

Function	Use to
@AddVertices	Interpolate new coordinates at specified intervals along a feature
@Affine	Apply an affine transformation to the current feature's coordinates
@Arc	Vectorize an arc or ellipse
@Bounds	Extract minimum & maximum bounds of a feature
@Buffer	Generate a buffer region around the current feature
@Close	Close a line feature to form a polygon
@ConvertToArc	Calculate the circular arc that passes through three specified points
@ConvertToLine	Replace the geometry of a feature with a line
@ConvertToPoint	Replace the geometry of a feature with a point
@ConvexHull	Replace the geometry of a feature with a convex hull
@Coordinate	Return the value of a single coordinate of the current feature
@CoordSys	Return the coordinate system of the current feature
@Dimension	Report or set the coordinate dimension of the current feature
@FitDGN	Move and scale the specified Design file to fit within the bounding box of the current feature (DGN)
@Force2D	Force the current feature to be 2D and drop the Z value(s)
@Generalize	Thin the points on a line or polygonal feature
@GeneratePoint	Generate a point inside a polygon, or donut polygon feature
@GeometryType	Return or set the geometry type of the current feature
@NumHoles	Return the number of holes in a donut polygon feature
@Offset	Add the offsets to the feature's coordinates
@Orient	Adjust the orientation of a polygonal feature or the direction of a line feature using left-hand or right-hand rules

Attributes

	Function	Use to
Math	@Split	Break a string into its component parts and assign those parts to attributes of the current feature
	@SupplyAttributes	Assign a value to an attribute, or pack all attribute names & values into a single string
	@Area	Calculate the area of a polygonal feature
	@Circularity	Compute a circularity measure for area-based features
	@ConvertBase	Convert a value to a new base
	@Evaluate	Evaluate an arithmetic expression and return the result
	@Length	Calculate the length or perimeter of the current feature
Miscellaneous	@TCL	Evaluate a Tool Command Language (TCL) expression and return the result.
	@Abort	Abort a data translation and log the current feature
	@CoordSys	Return the name of the coordinate system for the current feature
	@Count	Generate a unique number (count) in a named domain
	@File	Read or Write attribute values from or to a data file
	@Filenamepart	Extract part of a file name (directory, root or extension)
	@GlobalVariable	Set or read an FME global variable
	@GOID	Calculate a Geographic Object Identifier (GOID) for the current feature
	@Log	Output details of the current feature to the FME log file
	@NumCoords	Return the number of coordinates that define the current feature's geometry
	@SDEsql	Execute database commands through an SDE connection to the database from within FME
	@SQL	Execute database commands through a connection to an ODBC or ORACLE database
	@System	Execute any external command from within an FME mapping file
	@Timestamp	Return the current time in a specified format
	@Transform	Correlate features from the reader to writer format

Factories

	Factory	Use to
Geometry	AggregateFactory	Aggregate feature geometries into a single feature
	ArcFactory	Join features using common nodes and the same attribute values
	BoundingBoxFactory	Compute the 2D bounding box of a set of features
	ChoppingFactory	Break a feature into smaller features
	ClippingFactory	Clip features with a polygon, donut or aggregate feature
	CloseFactory	Create polygons from line features
	CommonSegmentFactory	Find features with any common line segment
	ConnectionFactory	Join features by connecting point or line features to create lines or polygons
	ConvexHullFactory	Compute a convex hull (enclosing convex polygon) from a set of input features
	CreationFactory	Create new feature with the specified geometry
	DeaggregateFactory	Break an aggregate feature into it's constituent parts
	DonutFactory	Create donut or PIP polygons from existing polygon, donut and point features
	DonutHoleFactory	Split a donut polygon into its constituent parts (outer shell polygon and hole polygons)
	ExtensionFactory	Extend a line feature by an extension length
	IntersectionFactory	Create new nodes where features intersect or cross
	LabelFactory	Create label points at specified intervals along line or polygon features
	MatchingFactory	Detect and flag features that have matching geometry and/or matching attribute values
	OverlayFactory	Detect which features overlap or intersect and merge their attributes
	PIPComponentsFactory	Split a point-in-polygon geometry into its constituent parts (outer polygon and point)
	PolygonDissolveFactory	Dissolve polygons into larger polygons by removing shared edges
PolygonFactory	Create polygons from line features	
ProximityFactory	Detect the closest feature to a base feature set or determine which features intersect or are enclosed by a single base feature	

	Factory	Use to
Attribute	ReferenceFactory	Resolve attributes references between features and merge their geometry or attributes
	SmallworldGeometryFactory	Split complex Smallworld geometries into their constituent features
	TilingFactory	Create aggregates from input features, grouping them into tiles covering the specified width and height
	TopologyFactory	Create topological information for line or polygon features
	WarpFactory	Perform warping on feature coordinates based on a set of reference features
	CorrelationFactory	Correlate feature attributes based on a database table
	ElementFactory	Create new features from an attribute list
Miscellaneous	ListFactory	Create an attribute list from the attributes of a set of input features
	MatchingFactory	Detect and flag features that have matching geometry and/or matching attribute values
	ReferenceFactory	Resolve attributes references between features and merge their geometry or attributes
	SortFactory	Sort features based on the values of a list of attributes
	BranchingFactory	Route features to a named factory
	OracleQueryFactory	Retrieve spatial data stored using the Oracle Spatial Cartridge, or Oracle 8i Spatial in either the relational or object modes
	RecorderFactory	Read or write features from/to an FME Feature Store File (FFS)
	ReportFactory	Report the attributes of a feature to an ASCII file
	SamplingFactory	Output every n^{th} feature (where n is the sample rate)
	SDEQueryFactory	Perform an ESRI SDE 2.1 spatial query
	SDE30QueryFactory	Perform an ESRI SDE 3.0 spatial query
	TeeFactory	Replicate each input feature and output zero or more copies
	TestFactory	Apply a test clause to a feature or its attributes and either pass or fail the feature

List of Directives

Directive Name	Directive Type	More Information
COORDINATE_SYSTEM_DEF	Coordinate System Definition	Search FME Fundamentals help.
DATUM_DEF	Local Coordinate System Definition – Datum	Search FME Fundamentals help.
DEFAULT_MACRO	Default Macro	Search FME Fundamentals help.
ELLIPSOID_DEF	Local Coordinate System Definition – Ellipsoid	Search FME Fundamentals help.
FME_ARC_DEGREES_PER_EDGE	Default arc edge angle (@Arc function)	@Arc section
FME_ARC_EDGE_TOLERANCE	Arc thinning tolerance (@Arc function)	@Arc section
FME_DEBUG	Debugging directive	Search FME Fundamentals help.
FME_DESKTOP	FME Desktop indicator, Predefined Macro	Search FME Fundamentals help.
FME_HOME	FME home directory, Predefined Macro	Search FME Fundamentals help.
FME_HOME_DOS	FME home directory, Predefined Macro	Search FME Fundamentals help.
FME_HOME_UNIX	FME home directory, Predefined Macro	Search FME Fundamentals help.
FME_MF_DIR	FME mapping file directory, Predefined Macro	Search FME Fundamentals help.
FME_MF_DIR_DOS	FME mapping file directory, Predefined Macro	Search FME Fundamentals help.
FME_MF_DIR_UNIX	FME mapping file directory, Predefined Macro	Search FME Fundamentals help.
FME_MF_NAME	FME mapping file name, Predefined Macro	Search FME Fundamentals help.
FME_MINIMUM_BUILD	Minimum Build Requirement	Search FME Fundamentals help.
GUI_CHOICE	User Interface Directive, chosen values for text and numeric parameters	Search FME Fundamentals help.
GUI_COORDSYS	User Interface Directive, coordinate system	Search FME Fundamentals help.
GUI_DIRNAME	User Interface Directive, directory name	Search FME Fundamentals help.
GUI_FILENAME	User Interface Directive, file name	Search FME Fundamentals help.

Directive Name	Directive Type	More Information
GUI_FLOAT	User Interface Directive, floating point	Search FME Fundamentals help.
GUI_INTEGER	User Interface Directive, integer	Search FME Fundamentals help.
GUI_LISTBOX	User Interface Directive, list	Search FME Fundamentals help.
GUI_PASSWORD	User Interface Directive, password	Search FME Fundamentals help.
GUI_TEXT	User Interface Directive, text	Search FME Fundamentals help.
GUI_TITLE	User Interface Directive, dialog box title	Search FME Fundamentals help.
INCLUDE	Include directive	Search FME Fundamentals help.
LOG_APPEND	Log file configuration	Search FME Fundamentals help.
LOG_FILENAME	Log file configuration	Search FME Fundamentals help.
LOG_MAX_FEATURES	Log file configuration	<ul style="list-style-type: none"> •@Log section •Search FME Fundamentals help.
LOG_STANDARDOUT	Log file configuration	Search FME Fundamentals help.
MACRO	Macro directive	Search FME Fundamentals help.
MAPPING_FILE_ID	Mapping File Identification directive	Search FME Fundamentals help.
READER_KEYWORD	Reader Keyword directive	Search FME Fundamentals help.
<READER_KEYWORD>_ COORDINATE_SYSTEM	Coordinate System Identification	Search FME Fundamentals help.
READER_TYPE	Reader directive	Search FME Fundamentals help.
UNIT_DEF	Local Coordinate System Definition – units	Search FME Fundamentals help.
WRITER_KEYWORD	Writer Keyword directive	Search FME Fundamentals help.
<WRITER_KEYWORD>_ COORDINATE_SYSTEM	Coordinate System Identification	Search FME Fundamentals help.
WRITER_TYPE	Writer directive	Search FME Fundamentals help.

List of Environment Variables

Environment Variable Name	Directive Type	More Information
FME_TEMP	FME Temporary file directory.	Search FME Fundamentals help.

Syntax of Tcl Regular Expressions

Note: FME uses Tcl version 8.4.12.

A regular expression describes strings of characters. It's a pattern that matches certain strings and doesn't match others.

Different Flavors of REs

Regular expressions (REs), as defined by POSIX, come in two flavors: extended REs (EREs) and basic REs (BREs).

- EREs are roughly those of the traditional `egrep`,
- BREs are roughly those of the traditional `ed`.

This implementation adds a third flavor – advanced REs (AREs) – which are basically EREs with some significant extensions.

This manual page primarily describes AREs. BREs mostly exist for backward compatibility in some old programs; they will be discussed at the end. POSIX EREs are almost an exact subset of AREs. Features of AREs that are not present in EREs will be indicated.

Regular Expression Syntax¹

An ARE is one or more branches, separated by `|`, matching anything that matches any of the branches.

A branch is zero or more constraints or quantified atoms, concatenated. It matches a match for the first, followed by a match for the second, etc; an empty branch matches the empty string.

A quantified atom is an atom possibly followed by a single quantifier. Without a quantifier, it matches a match for the atom. The quantifiers, and what a so-quantified atom matches, are:

* a sequence of 0 or more matches of the atom

1. Tcl regular expressions are implemented using the package written by Henry Spencer, based on the 1003.2 spec and some (not quite all) of the Perl5 extensions. Much of the description of regular expressions below is copied verbatim from his manual entry.

<code>+</code>	a sequence of 1 or more matches of the atom
<code>?</code>	a sequence of 0 or 1 matches of the atom
<code>{m}</code>	a sequence of exactly <i>m</i> matches of the atom
<code>{m, }</code>	a sequence of <i>m</i> or more matches of the atom
<code>{m, n}</code>	a sequence of <i>m</i> through <i>n</i> (inclusive) matches of the atom; <i>m</i> may not exceed <i>n</i>
<code>*? +? ?? {m}? {m, }? {m, n}?</code>	non-greedy quantifiers, which match the same possibilities, but prefer the smallest number rather than the largest number of matches (see <i>MATCHING</i>)

The forms using `{` and `}` are known as bounds. The numbers *m* and *n* are unsigned decimal integers with permissible values from 0 to 255 inclusive.

An atom is one of:

<code>(re)</code>	(where <i>re</i> is any regular expression) matches a match for <i>re</i> , with the match noted for possible reporting
<code>(?:re)</code>	as previous, but does no reporting (a ``non-capturing'' set of parentheses)
<code>()</code>	matches an empty string, noted for possible reporting
<code>(?:)</code>	matches an empty string, without reporting
<code>[chars]</code>	a bracket expression, matching any one of the chars (see <i>BRACKET EXPRESSIONS</i> for more detail)
<code>.</code>	matches any single character
<code>\k</code>	(where <i>k</i> is a non-alphanumeric character) matches that character taken as an ordinary character, e.g. <code>\\</code> matches a backslash character
<code>\c</code>	where <i>c</i> is alphanumeric (possibly followed by other characters), an escape (AREs only), see <i>ESCAPES</i> below
<code>{</code>	when followed by a character other than a digit, matches the left-brace character <code>`{'</code> ; when followed by a digit, it is the beginning of a bound (see above)
<code>x</code>	where <i>x</i> is a single character with no other significance, matches that character.

A constraint matches an empty string when specific conditions are met. A constraint may not be followed by a quantifier. The simple constraints are as follows; some more constraints are described later, under *ESCAPES*.

<code>^</code>	matches at the beginning of a line
<code>\$</code>	matches at the end of a line
<code>(?=re)</code>	positive lookahead (AREs only), matches at any point where a substring matching <i>re</i> begins
<code>(?!re)</code>	negative lookahead (AREs only), matches at any point where no substring matching <i>re</i> begins

The lookahead constraints may not contain back references (see later), and all parentheses within them are considered non-capturing.

An RE may not end with \.

BRACKET EXPRESSIONS A bracket expression is a list of characters enclosed in []. It normally matches any single character from the list (but see below). If the list begins with ^, it matches any single character (but see below) not from the rest of the list.

If two characters in the list are separated by -, this is shorthand for the full range of characters between those two (inclusive) in the collating sequence, e.g. [0-9] in ASCII matches any decimal digit. Two ranges may not share an endpoint, so for example, a-c-e is illegal. Ranges are very collating-sequence-dependent, and portable programs should avoid relying on them.

To include a literal] or - in the list, the simplest method is to enclose it in [. and .] to make it a collating element (see below). Alternatively, make it the first character (following a possible ^), or (AREs only) precede it with \. Alternatively, for -, make it the last character, or the second endpoint of a range. To use a literal - as the first endpoint of a range, make it a collating element or (AREs only) precede it with \. With the exception of these, some combinations using [(see next paragraphs), and escapes, all other special characters lose their special significance within a bracket expression.

Within a bracket expression, a collating element (a character, a multi-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in [. and .] stands for the sequence of characters of that collating element. The sequence is a single element of the bracket expression's list. A bracket expression in a locale that has multi-character collating elements can thus match more than one character. | So (insidiously), a bracket expression that starts with ^| can match multi-character collating elements even if none | of them appear in the bracket expression! (Note: Tcl currently has no multi-character collating elements. This information is only for illustration.)

For example, assume the collating sequence includes a ch | multi-character collating element. Then the RE [[.ch.]]*c | (zero or more ch's followed by c) matches the first five | characters of `chchcc`. Also, the RE [^c]b matches all of | `chb` (because [^c] matches the multi-character ch).

Within a bracket expression, a collating element enclosed in [= and =] is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were `[' and `']`.) For example, if o and ^ are the members of an equivalence class, then `[[=o=]]`, `[[=^=]]`, and `[o^]` are all synonymous. An equivalence class may not be an endpoint of a range. (Note: Tcl currently implements only the Unicode | locale. It doesn't define any equivalence classes. The | examples above are just illustrations.)

Within a bracket expression, the name of a character class enclosed in `[:` and `:]` stands for the list of all characters (not all collating elements!) belonging to that class. Standard character classes are:

- `alpha` A letter – uppercase letter
- `lower` A – lowercase letter
- `digit` – decimal digit
- `xdigit` – hexadecimal digit
- `alnum` – alphanumeric (letter or digit)
- `print` – An alphanumeric (same as `alnum`)
- `blank` – space or tab character
- `space` – character producing white space in displayed text
- `punct` – punctuation character
- `graph` – character with a visible representation
- `cntrl` – control character

A locale may provide others. (Note that the current Tcl implementation has only one locale: the Unicode locale.) A character class may not be used as an endpoint of a range.

There are two special cases of bracket expressions: the bracket expressions `[[:<:]]` and `[[:>:]]` are constraints, matching empty strings at the beginning and end of a word respectively. A word is defined as a sequence of word characters that is neither preceded nor followed by word characters. A word character is an `alnum` character or an underscore (`_`). These special bracket expressions are deprecated; users of AREs should use constraint escapes instead (see below).

ESCAPES Escapes (AREs only), which begin with a `\` followed by an alphanumeric character, come in several varieties: character entry, class shorthands, constraint escapes, and back references. A `\` followed by an alphanumeric character but not constituting a valid escape is illegal in AREs. In EREs, there are no escapes: outside a bracket expression, a `\` followed by an alphanumeric character merely stands for that character as an ordinary character, and inside a bracket expression, `\` is an ordinary character. (The latter is the one actual incompatibility between EREs and AREs.)

Character-entry escapes (AREs only) exist to make it easier to specify non-printing and otherwise inconvenient characters in REs:

<code>\a</code>	alert (bell) character, as in C
<code>\b</code>	backspace, as in C
<code>\B</code>	synonym for <code>\</code> to help reduce backslash doubling in some applications where there are multiple levels of backslash processing
<code>\cX</code>	(where X is any character) the character whose low-order 5 bits are the same as those of X, and whose other bits are all zero
<code>\e</code>	the character whose collating-sequence name is <code>`ESC'</code> , or failing that, the character with octal

```

                                value 033
\f          formfeed, as in C
\n          newline, as in C
\r          carriage return, as in C
\t          horizontal tab, as in C
\uvwxyz    (where wxyz is exactly four hexadecimal digits) the
            Unicode character U+wxyz in the local byte ordering
\Ustuvwxyz where stuvwxyz is exactly eight hexadecimal digits)
            reserved for a somewhat-hypothetical Unicode exten-
            sion to 32 bits
\v          vertical tab, as in C are all available.
\xhhh      (where hhh is any sequence of hexadecimal digits)
            the character whose hexadecimal value is 0xhhh (a
            single character no matter how many hexadecimal
            digits are used).
\0          the character whose value is 0
\xy        (where xy is exactly two octal digits, and is not a
            back reference (see below)) the character whose
            octal value is 0xy
\xyz       (where xyz is exactly three octal digits, and is
            not a back reference (see below)) the character
            whose octal value is 0xyz

```

Hexadecimal digits are '0'-'9', 'a'-'f', and 'A'-'F'. Octal digits are '0'-'7'.

The character-entry escapes are always taken as ordinary characters. For example, `\135` is `]` in ASCII, but `\135` does not terminate a bracket expression. Beware, however, that some applications (e.g., C compilers) interpret such sequences themselves before the regular-expression package gets to see them, which may require doubling (quadrupling, etc.) the `\`.

Class-shorthand escapes (AREs only) provide shorthands for certain commonly-used character classes:

```

\d          [[:digit:]]
\s          [[:space:]]
\w          [[:alnum:]]_ (note underscore)
\D          [^[:digit:]]
\S          [^[:space:]]
\W          [^[:alnum:]]_ (note underscore)

```

Within bracket expressions, `\d`, `\s`, and `\w` lose their outer brackets, and `\D`, `\S`, and `\W` are illegal. (So, for example, `[a-c\d]` is equivalent to `[a- | c[:digit:]]`. Also, `[a-c\D]`, which is equivalent to `[a- | c^[:digit:]]`, is illegal.)

A constraint escape (AREs only) is a constraint, matching the empty string if specific conditions are met, written as an escape:

```

\A          matches only at the beginning of the string (see
            MATCHING, below, for how this differs from `^')
\m          matches only at the beginning of a word
\M          matches only at the end of a word

```

<code>\y</code>	matches only at the beginning or end of a word
<code>\Y</code>	matches only at a point that is not the beginning or end of a word
<code>\Z</code>	matches only at the end of the string (see <i>MATCHING</i> , below, for how this differs from <code>`\$'</code>)
<code>\m</code>	(where <i>m</i> is a nonzero digit) a back reference, see below
<code>\mnn</code>	(where <i>m</i> is a nonzero digit, and <i>nn</i> is some more digits, and the decimal value <i>mnn</i> is not greater than the number of closing capturing parentheses seen so far) a back reference, see below

A word is defined as in the specification of `[[:<:]]` and `[[:>:]]` above. Constraint escapes are illegal within bracket expressions.

A back reference (AREs only) matches the same string matched by the parenthesized subexpression specified by the number, so that (e.g.) `((bc))1` matches `bb` or `cc` but not ``bc'`. The subexpression must entirely precede the back reference in the RE. Subexpressions are numbered in the order of their leading parentheses. Non-capturing parentheses do not define subexpressions.

There is an inherent historical ambiguity between octal character-entry escapes and back references, which is resolved by heuristics, as hinted at above. A leading zero always indicates an octal escape. A single non-zero digit, not followed by another digit, is always taken as a back reference. A multi-digit sequence not starting with a zero is taken as a back reference if it comes after a suitable subexpression (i.e. the number is in the legal range for a back reference), and otherwise is taken as octal.

METASYNTAX In addition to the main syntax described above, there are some special forms and miscellaneous syntactic facilities available.

Normally the flavor of RE being used is specified by application-dependent means. However, this can be overridden by a director. If an RE of any flavor begins with ``***:'`, the rest of the RE is an ARE. If an RE of any flavor begins with ``***='`, the rest of the RE is taken to be a literal string, with all characters considered ordinary characters.

An ARE may begin with embedded options: a sequence `(?xyz)` (where *xyz* is one or more alphabetic characters) specifies options affecting the rest of the RE. These supplement, and can override, any options specified by the application. The available option letters are:

<code>b</code>	rest of RE is a BRE
<code>c</code>	case-sensitive matching (usual default)
<code>e</code>	rest of RE is an ERE
<code>i</code>	case-insensitive matching (see <i>MATCHING</i> , below)
<code>m</code>	historical synonym for <code>n</code>
<code>n</code>	newline-sensitive matching (see <i>MATCHING</i> , below)
<code>p</code>	partial newline-sensitive matching (see <i>MATCHING</i> , below)

q	rest of RE is a literal ("quoted") string, all ordinary characters
s	non-newline-sensitive matching (usual default)
t	tight syntax (usual default; see below)
w	inverse partial newline-sensitive ("weird") matching (see <i>MATCHING</i> , below)
x	expanded syntax (see below)

Embedded options take effect at the `)` terminating the sequence. They are available only at the start of an ARE, and may not be used later within it.

In addition to the usual (tight) RE syntax, in which all characters are significant, there is an expanded syntax, available in all flavors of RE with the `-expanded` switch, or in AREs with the embedded `x` option. In the expanded syntax, white-space characters are ignored and all characters between a `#` and the following newline (or the end of the RE) are ignored, permitting paragraphing and commenting a complex RE. There are three exceptions to that basic rule:

- a white-space character or `#` preceded by `\` is retained
- white space or `#` within a bracket expression is retained
- white space and comments are illegal within multi-character symbols like the ARE `(?:` or the BRE `\(`

Expanded-syntax white-space characters are blank, tab, newline, and any character that belongs to the space character class.

Finally, in an ARE, outside bracket expressions, the sequence `(?#ttt)` (where `ttt` is any text not containing a `)`) is a comment, completely ignored. Again, this is not allowed between the characters of multi-character symbols like `(?:`. Such comments are more a historical artifact than a useful facility, and their use is deprecated; use the expanded syntax instead.

None of these metasyntax extensions is available if the application (or an initial `***=` director) has specified that the user's input be treated as a literal string rather than as an RE.

MATCHING In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, its choice is determined by its preference: either the longest substring, or the shortest.

Most atoms, and all constraints, have no preference. A parenthesized RE has the same preference (possibly none) as the RE. A quantified atom with quantifier `{m}` or `{m}?` has the same preference (possibly none) as the atom itself. A quantified atom with other normal quantifiers (including `{m,n}` with `m` equal to `n`) prefers longest match. A quantified atom with other non-greedy quantifiers (including `{m,n}? with m equal to n) prefers shortest match. A branch has the same preference as the first quantified atom in it which has a preference. An RE`

consisting of two or more branches connected by the `|` operator prefers longest match.

Subject to the constraints imposed by the rules for matching the whole RE, subexpressions also match the longest or shortest possible substrings, based on their preferences, with subexpressions starting earlier in the RE taking priority over ones starting later. Note that outer subexpressions thus take priority over their component subexpressions.

Note that the quantifiers `{1,1}` and `{1,1}?` can be used to force longest and shortest preference, respectively, on a subexpression or a whole RE.

Match lengths are measured in characters, not collating elements. An empty string is considered longer than no match at all. For example, `bb*` matches the three middle characters of `'abbbc'`, `(week|wee)(night|knights)` matches all ten characters of `'weeknights'`, when `(.*)` is matched against `abc` the parenthesized subexpression matches all three characters, and when `(a*)` is matched against `bc` both the whole RE and the parenthesized subexpression match an empty string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, so that `x` becomes `[xX]`. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, so that `[x]` becomes `[xX]` and `[^x]` becomes `[^xX]`.

If newline-sensitive matching is specified, `.` and bracket expressions using `^` will never match the newline character (so that matches will never cross newlines unless the RE explicitly arranges it) and `^` and `$` will match the empty string after and before a newline respectively, in addition to matching at beginning and end of string respectively. ARE `\A` and `\Z` continue to match beginning or end of string only.

If partial newline-sensitive matching is specified, this affects `.` and bracket expressions as with newline-sensitive matching, but not `^` and `$`.

If inverse partial newline-sensitive matching is specified, this affects `^` and `$` as with newline-sensitive matching, but not `.` and bracket expressions. This isn't very useful but is provided for symmetry.

LIMITS AND COMPATIBILITY No particular limit is imposed on the length of REs. Programs intended to be highly portable should not employ REs longer than 256 bytes, as a POSIX-compliant implementation can refuse to accept such REs.

The only feature of AREs that is actually incompatible with POSIX EREs is that `\` does not lose its special significance inside bracket expressions. All other ARE

features use syntax which is illegal or has undefined or unspecified effects in POSIX EREs; the `***` syntax of directors likewise is outside the POSIX syntax for both BREs and EREs.

Many of the ARE extensions are borrowed from Perl, but some have been changed to clean them up, and a few Perl extensions are not present. Incompatibilities of note include `\b`, `\B`, the lack of special treatment for a trailing newline, the addition of complemented bracket expressions to the things affected by newline-sensitive matching, the restrictions on parentheses and back references in lookahead constraints, and the longest/shortest- match (rather than first-match) matching semantics.

The matching rules for REs containing both normal and non- greedy quantifiers have changed since early beta-test versions of this package. (The new rules are much simpler and cleaner, but don't work as hard at guessing the user's real intentions.)

Henry Spencer's original 1986 regexp package, still in widespread use (e.g., in pre-8.1 releases of Tcl), implemented an early version of today's EREs. There are four incompatibilities between regexp's near-EREs ('RREs' for short) and AREs. In roughly increasing order of significance:

In AREs, `\` followed by an alphanumeric character is either an escape or an error, while in RREs, it was just another way of writing the alphanumeric. This should not be a problem because there was no reason to write such a sequence in RREs.

`{` followed by a digit in an ARE is the beginning of a bound, while in RREs, `{` was always an ordinary character. Such sequences should be rare, and will often result in an error because following characters will not look like a valid bound.

In AREs, `\` remains a special character within `[]`, so a literal `\` within `[]` must be written `\\`. `\\` also gives a literal `\` within `[]` in RREs, but only truly paranoid programmers routinely doubled the backslash.

AREs report the longest/shortest match for the RE, rather than the first found in a specified search order. This may affect some RREs which were written in the expectation that the first match would be reported. (The careful crafting of RREs to optimize the search order for fast matching is obsolete (AREs examine all possible matches in parallel, and their performance is largely insensitive to their complexity) but cases where the search order was exploited to deliberately find a match which was not the longest/shortest will need rewriting.)

BASIC REGULAR EXPRESSIONS BREs differ from EREs in several respects. `|`, `+`, and `?` are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are `\{` and `\}`, with `{` and `}` by themselves ordinary characters. The parentheses for nested subexpressions are `\(` and `\)`, with `(` and `)` by themselves ordinary characters. `^` is an ordinary character except at the beginning of the RE or the beginning of a parenthesized

subexpression, \$ is an ordinary character except at the end of the RE or the end of a parenthesized subexpression, and * is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading `^'). Finally, single-digit back references are available, and \< and \> are synonyms for [[:<:]] and [[:>:]] respectively; no other escapes are available.