INTERNATIONAL STANDARD

**ISO 19107**

Second edition
2019-12

# Geographic information — Spatial schema

*Information géographique — Schéma spatial*



Reference number
ISO 19107:2019(E)



ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

# Contents

Page

# Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing documents is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/TC 211, *Geographic information/Geomatics*.

This second edition cancels and replaces the first edition (ISO 19107:2003), which has been technically revised. The main changes compared to the previous edition are as follows:

— It now forms a logical subset of this second edition. In other words, this document is 100 % backwardly compatible with its previous version, ISO 19107:2003, except in a few areas (in NURBS) where the previous version contained technical errors that are corrected in this revision.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

# Introduction

This document provides conceptual schemas for describing, representing and manipulating the spatial characteristics of geographic entities. Standardization in this area is the cornerstone for other geographic information design, specification and standardization.

"Vector" data consists of geometric primitives used to construct expressions of the spatial characteristics of geographic features. "Raster" data is based on the division of the extent covered into small units according to a tessellation of the space. This document deals only with vector data.

There is a hierarchy of complexity in the "geometry" of the underlying object used in various coordinate systems. These may use reference planes (map geometry – Euclidean), reference spheres (spherical geometry — using spherical trigonometry), reference ellipsoids (ellipsoidal geometry using Gaussian or Riemannian metrics) or more complex surfaces (usually using numeric approximations for calculation). The coordinates of a point locate it on, or in relation to, the reference geometry. With the exception of "map geometry," the usual Euclidean formulae for distance and area do not apply directly in the coordinate system.

Topology expressions provide qualitative descriptions of the spatial relations between geometry objects. Topology deals with the characteristics of geometric figures that remain invariant if the space is deformed elastically. Topological properties do not change when information is transformed from one coordinate system to another, usually including the coordinate function that map from R2 or R3 to the reference geometry. Topological properties in the domain of the coordinate system will be identical to those on the geographic surface; but the metric properties may change significantly (e.g. distance, area, direction).

Spatial operators are functions and procedures that use, query, create, modify or delete spatial objects. This document defines the taxonomy of some of the more important operators, their definitions and implementations. The goals are to:

— Define spatial operators unambiguously, so that different implementations will yield comparable results within the limitations of accuracy and resolution.

— Use these definitions to define a set of standard operations that will form the basis of compliant systems and thus act as a test-bed for implementers and a benchmark set for validation of compliance.

— Define an operator algebra that will allow combinations of the base operators to be used predictably in the query and manipulation of geographic feature data.

Standardized conceptual schemas for spatial characteristics will increase the ability to share geographic information between applications. These schemas will be used by geographic information system and software developers and users of geographic information to provide consistently understandable spatial data structures and functions.

This document is technical because geometry is a technical topic. Euclid was speaking of a simpler form of geometry to the most powerful man in his world when he said:

*There is no royal road to geometry (μή εἶναι βασιλικήν ἀτραπόν ἐπί γεωμετρίαν).*

*Euclid to Ptolemy I Soter (General with Alexander the Great, Pharaoh of Egypt) —*

*Attributed by Proclus (412–485 AD) in Commentary on the First Book of Euclid's Elements*

# Geographic information — Spatial schema

## 1 Scope

This document specifies conceptual schemas for describing the spatial characteristics of geographic entities, and a set of spatial operations consistent with these schemas. It treats "vector" geometry and topology. It defines standard spatial operations for use in access, query, management, processing and data exchange of geographic information for spatial (geometric and topological) objects. Because of the nature of geographic information, these geometric coordinate spaces will normally have up to three spatial dimensions, one temporal dimension and any number of other spatially dependent parameters as needed by the applications. In general, the topological dimension of the spatial projections of the geometric objects will be at most three.

## 2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 19103, *Geographic information — Conceptual schema language*

ISO 19108, *Geographic information — Temporal schema*

ISO 19109, *Geographic information — Rules for application schema*

ISO 19111, *Geographic information — Spatial referencing by coordinates*

ISO/IEC 11404:2007, *Information technology — General-Purpose Datatypes (GPD)*

ISO/IEC 19505-2:2012, *Information technology — Object Management Group Unified Modeling Language (OMG UML) — Part 2: Superstructure*

## 3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 11404, ISO 19103, ISO/IEC 19505-2 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

NOTE Common words from geometry, such as point, curve, line, surface solid, etc., take the common meanings unless they are used as classifier names (usually interfaces), in which case they are a digital representation of the geometric concept. Common mathematical terms that are not defined here take on their common meanings in mathematics (see [15], [10], ISO/IEC 11404 or a standard text on the topic, such as the "N. Bourbaki"[1)] series currently published by Springer Verlag, in French, English and German). Care should be taken since mathematical terms can be context sensitive, and can easily be confused with common words. For example, "open" set, "closed" curve, "closed" set, "rational" function, "boundary," "interior," "closure", "exterior", "function" and others from common language but have very specific meanings in mathematics and in this document. Where necessary to prevent confusion, existing definitions have been elaborated to make their intent in this document explicit. Mathematical terms include common vocabulary from geometry, topology, calculus, geodesy and differential geometry. Many of these terms can be sufficiently common that inclusion is not necessary. They are included here to prevent confusion especially for terms like the ones listed above that have another meaning in another context.

**3.1**
**abstract root**
<programming> common root classifier of a category which is a superclass of any other classifier in the category

Note 1 to entry: The class Any in some programming languages is the abstract root of all classes. Thus, it is the *de facto* union of all classes. In this document, Geometry is the (named and explicit) abstract root for all geometry objects. In the package Geometry and any of its subpackages (including those in its Requirements Classes), any interface will be a subtype of Geometry either directly or transitively.

**3.2**
**arc**
<geometry> *segment* (3.83) of a curve

**3.3**
**barycentric coordinates**
<coordinate geometry> point in a **n-dimension** coordinate system using n+1 numbers, $[u_0, u_1, u_2, u_3, \ldots, u_n] \ni [[0 \le u_i \le 1] \wedge \sum u_i = 1,0]$, in which the location of a point of an **n-simplex** (of any dimension) is specified by a weighted centre of mass of equal masses placed at its vertices using vector algebra of the $\mathbb{R}^n$ used in the **coordinate reference system**

Note 1 to entry: Even though there are n+1 coordinates in a barycentric coordinate system, the topological dimension is n, since the restriction (sums to 1,0) loses 1 degree of freedom (once you have n ordinates, the remaining one is determined such as $u_n = 1,0 - \sum_{i=0}^{n-1} u_i$ . The coordinates for the simplex are all non-negative, but the system can be extended outside of the simples by using negative numbers. If the ordinates are all positive, then the point is inside (interior to) the **n-simplex**. If one of them is 1,0 and the other 0, this is a corner of the simplex. If one of them is zero and the others still each greater than or equal to zero, the point is on the n−1-simplex opposite the vertex zeroed out. If any are negative, the point is outside of the simplex. The coordinates are dependent on the underlying coordinate reference system of the source data.



1) N. Bourbaki is the pen-name for the "Association des collaborateurs de Nicolas Bourbaki" (Association of Collaborators of Nicolas Bourbaki) of mathematicians first published in 1935 and dedicated to "formalizing" mathematics. The group has an office at the "École Normale Supérieure" in Paris. See https://en.wikipedia.org/wiki/Nicolas_Bourbaki. Their books are held in high regard by the mathematical community.

**3.4**
**bearing**
horizontal angle, tangent or direction at a point

Note 1 to entry: This definition (as opposed to the one in ISO 19162:2015) is required for this document because the concept is used in other definitions, such as first geodetic problem and second geodetic problem. The two definitions are nearly equivalent because the tangent of a curve on a surface is a tangent to the surface and does specify a direction. Usual 2D measure of bearing can be an angle equivalently measured from North clockwise, or a unit tangent vector. If the coordinate system is spatially 3D, the horizontal bearing angle may also need to a vertical altitude angle to be complete. If a reference curve (as used in ISO 19162) is parameterized by arc length, then the "derivative" is a unit vector. If another parameterization "$t$" is used, then the derivative should be normalized $\left(\vec{\tau} / \left\|\vec{\tau}\right\|; \dot{c}(t) = \vec{\tau};\right)$. This is useful, since parameterization by arc length can be computationally difficult. The numeric representation of a vector depends on the coordinate system. The bearing is not dependent on a coordinate system, but it can be represented in any reasonable system. The bearing is not dependent on its various representations.

**3.5**
**bicontinuous**
<mathematics> invertible, continuous and with a continuous inverse

**3.6**
**boundary**
set that represents the limit of an entity

Note 1 to entry: Boundary is most commonly used in the context of geometry, where the set is a collection of points or a collection of objects that represent those points. In other arenas, the term is used metaphorically to describe the transition between an entity and the rest of its domain of discourse.

**3.7**
**buffer**
geometric object containing all points and only those points whose distance from a specified geometric object is less than or equal to a given distance use in its construction

**3.8**
**closure**
union of the interior and boundary of a topological object or geometric object

**3.9**
**coboundary**
set of topological primitives of higher topological dimension associated with a particular topological object, such that this topological object is in each of their boundaries

Note 1 to entry: If a node is on the boundary of an edge, that edge is on the coboundary of that node. Any orientation parameter associated with one of these relations would also be associated with the other. The coboundary of a node can be called a "node star".

**3.10**
**conformal**, adj.
angle-preserving

Note 1 to entry: Some projections are conformal. For example, a Mercator preserves the angle between curves, so that if two curves in a Mercator projected plane cross at a 90°, then the preimage curves on the ellipsoid also cross at 90°, such as lines of constant latitude and lines of constant longitude.

**3.11**
**connected**
property of a topological space implying that only the entire space or the empty set are the only subsets which are both open and closed

Note 1 to entry: The formal definition of connected is that any pair of locally open sets whose union is the entire space must have a non-empty intersection.

a topological space T is connected if and only if

$$[\forall X, Y \subset T \ni X \cup Y = T] \Rightarrow [X \cap Y \neq \varnothing] \tag{1}$$

This formal definition is difficult to use. The term *path connected* (3.75), defined below is equivalent for the purposed of this document. The use or "finite precision" coordinates makes sets which are connected but not path connected impossible to represent. In all cases "connected" is used, but "path connected" is easier to test and to visualize.

**3.12**
**connected node**
<topology> node that starts or ends one or more edges

**3.13**
**control point**
<coordinate geometry> point used in the construction of a geometry that partially controls its shape but does not necessarily lie on the geometry

Note 1 to entry: A centre of an arc is a control point; poles in b-spline curves are control points.

**3.14**
**convex**
<geometry> containing all points on a "line" joining two interior points

Note 1 to entry: The definition of convex requires a definition of line. For coordinate systems, this is the usual linear interpolated arc, but in context, the "line" on a geometric reference surface will be a "geodesic arc". The default in this document is the linear interpolate.

**3.15**
**convex hull**
<geometry> smallest convex set containing a given geometric object

Note 1 to entry: "Smallest" is the set theoretic smallest, not an indication of a measurement. The definition can be rewritten as "the intersection of all convex sets that contain the geometric object". Another definition in a Euclidean space $\mathbb{E}^n$ is the union of all lines with both end points in the given geometric object.

$$C = A.convexHull \Leftrightarrow$$
$$[C.convex = TRUE] \wedge [A \subset C] \wedge [[B.convex = true, A \subset B] \Rightarrow [A \subseteq C \subseteq B]] \tag{2}$$

**3.16**
**coordinate**
one of a sequence of numbers designating the position of a point

Note 1 to entry: In a coordinate reference system, the numbers are qualified by units. The number of offsets (generally called "ordinates") in a coordinate is not the dimension. If there is a constraint, the dimension can be smaller. See *coordinate dimension* (3.17).

**3.17**
**coordinate dimension**
<coordinate geometry> number of separate decisions needed to describe a position in a coordinate system

Note 1 to entry: The coordinate dimension represents the number of choices made, and constraints can restrict choices. A barycentric coordinate which has (n+1)-offsets, but the underlying space is dimension n. Homogeneous coordinates (*wx*, *wy*, *wz*, *w*) are actually 3 dimensional because the choice of "*w*" does not affect the position, i.e. (wx, wy, wz, w) = (x y, z, 1) → (x, y, z) which is not affected by w. The dimension will be at most the count of the numbers in the coordinate, but it can be less if the coordinates are constrained in some manner.

**3.18**
**coordinate reference system**
<differential geometry, geodesy> coordinate system that is related to an object by a datum

[SOURCE: ISO 19111:2007, 4.8, modified — Notes 1 and 2 to entry have been added]

Note 1 to entry: The original definition of coordinate reference system (CRS) uses a geometric object (a geodetic datum) which is referenced to the real world by a Datum. ISO 19111 can extend this to any "parameter" which essentially can be represented as the graph of the parameter relation. This graph is in the cross product of the Datum, the domain space of the parameter function, and the parameter space (which in turn may be multi dimensional). Thus, every coordinate system used in this document is logically a CRS. Since this was not the original intent of ISO 19111, this document will use CRS for coordinate systems associated to a geodetic datum, and use the more general term "coordinate system" for anything else that is either more or less. Here, CRS means spatial coordinates. In every case where the underlying datum (surface) is not flat, the coordinate system is not Euclidean, and the metric is not Pythagorean.

Note 2 to entry: In the event that later versions of ISO 19111 change the definition as copied above, this document should not be affected. In the case that dynamic datums are used, the measurements made by the geometry operations of the objects in this document would only be valid at the time used for the datum and the associated coordinates. If the datum varies, the measure may also vary, but the definitions of the measures remains the same, but the actual values may be a function of time.

**3.19**
**curvature vector**
<differential geometry> second derivative of a curve parameterized by arc length, at a point

Note 1 to entry: If $c(s) = (x(s), y(s), z(s))$ is a curve in 3D Cartesian space ($\mathbb{E}^3$), and $s$ is the arc length along $c(s)$, then the unit tangent vector is $\dot{c}(s) = (\dot{x}(s), \dot{y}(s), \dot{z}(s))$, i.e. the derivative of the coordinate values of "$c$" with respect to "$s$". The curvature vector is $\ddot{c}(s) = (\ddot{x}(s), \ddot{y}(s), \ddot{z}(s))$. The curvature vector can be approximated by the inverse of the radius of a circle through any 3 nearby points on the curve (pointed from the curve to towards the centre of the circle)

**3.20**
**cycle**
<geometry, topology> bounded spatial object with an empty boundary

Note 1 to entry: Cycles are used to describe boundary components. A cycle usually has no boundary because it closes on itself, but it is bounded (i.e., it does not have infinite extent). A circle or a sphere, for example, has no boundary (i.e., its boundary is empty), but is bounded.

**3.21**
**data point**
<coordinate geometry> point that lies on the geometry

Note 1 to entry: The vertices in a line string are data points, the points used to construct a polynomial spline are data points. Data points can be used as control points, but are often derived after the geometry is constructed.

**3.22**
**diameter**
<metric> maximum distance between two points in the set of points

**3.23**
**directed edge**
<topology> directed topological object that represents an association between an edge and one of its orientations

Note 1 to entry: A directed edge that is in agreement with the orientation of the edge has a + orientation, otherwise, it has the opposite (–) orientation. Directed edge is used in topology to distinguish the right side (–) from the left side (+) of the same edge and the start node (–) and end node (+) of the same edge and in computational topology to represent these concepts.

**3.24**
**directed face**
<topology> directed topological object that represents an association between a face and one of its orientations

Note 1 to entry: The orientation of the directed edges that compose the exterior boundary of a directed face will appear positive from the direction of this vector; the orientation of a directed face that bounds a topological solid will point away from the topological solid. Adjacent solids would use different orientations for their shared boundary, consistent with the same sort of association between adjacent faces and their shared edges. directed faces are used in the coboundary relation to maintain the spatial association between face and edge.

**3.25**
**directed node**
<topology> directed topological object that represents an association between a node and one of its orientations

Note 1 to entry: Directed nodes are used in the coboundary relation to maintain the spatial association between edge and node. The orientation of a node is with respect to an edge, "+" for end node, "–" for start node. This is consistent with the vector notion of "result = end – start".

**3.26**
**directed solid**
<topology> directed topological object that represents an association between a topological solid and one of its orientations

Note 1 to entry: Directed solids are used in the coboundary relation to maintain the spatial association between face and topological solid. The orientation of a solid is with respect to a face, "+" if the upNormal is outward, "–" if inward. This is consistent with the concept of "up = outward" for a surface bounding a solid.

**3.27**
**distance**
<geometry, metric spaces> minimal length of a curve that joins the two points or geometries

Note 1 to entry: The usual distance function for two points in a coordinate space assumes an underlying plane and is a Euclidean distance. If the underlying Reference Surface is not a plane, then distance is defined by this minimum length of all curves between the two points. These surfaces are prime examples of non-Euclidean geometry, where the parallel postulate in Euclid's Elements does not hold. In mathematical terms, distance is the "greatest lower bound" of the length of the curves. The word minimum is sometimes used, but there should be no expectation that an instance of that minimum actually occurs, only that any larger number will have a length in the set that is smaller.

**3.28**
**domain**
<mathematics> well-defined set

Note 1 to entry: Domains are used to define the "domain and range" of operators and functions. If a function "f" maps values from the source domain "X" to values in the target domain "Y," then the notation is "f:X→Y." The relationship between the function and its source domain is referred to as its "domain." The relationship between the function and its target domain is referred to as its "range", "image" or alternatively "codomain". See the definition of "function" and "mapping" in [15].

**3.29**
**edge**
<topology> 1-dimensional topological primitive

Note 1 to entry: The geometric realization of an edge is a curve. The boundary of an edge is the set of one or two nodes associated with the edge within a topological complex.

**3.30**
**edge-node graph**
<topology> graph embedded within a topological complex composed of all of the edges and connected nodes within that complex

Note 1 to entry: The edge-node graph is a subcomplex of the complex within which it is embedded.

**3.31**
**ellipsoid**
<geodesy> geometric reference surface embedded in 3D Euclidean space represented by an ellipsoid of revolution where the rotation is about the polar axis

Note 1 to entry: For the Earth the rotation is about the polar axis. This results in an oblate ellipsoid with midpoint of the foci located at the nominal centre of the Earth.

Note 2 to entry: The two usual algorithms for latitude on an ellipsoid and on a sphere (such as used in spherical coordinates) are only equivalent if the ellipsoid is a sphere, having all radii equal in all directions. The problem is that a radial line from the centre of a general ellipsoid does not always cross the surface of the ellipsoid orthogonally. In general, planar slices through the centre do not intersect the surface orthogonally, and therefore the curves that correspond to the great circles of a sphere are not geodesics on the ellipsoid.

Note 3 to entry: The topology of the ellipsoid is inherited from the $\mathbb{E}^3$ space in which it is embedded. The difference is that metrics such as distance and direction on the ellipsoid are restricted to curves wholly on the surface and vectors tangent to the surface.

**3.32**
**empty set**
$\varnothing$
<mathematics> set without any elements

Note 1 to entry: Sets are equal if they contain exactly the same elements. Since any two empty sets would share exactly the same contained elements (by definition none), they are, by definition, equal. The empty set (∅) can be considered a geometric entity, because all the elements it contains are points. This is a vacuous statement since the set ∅ contains no elements, and therefore the "for all" statement has nothing to test and is thus true in each of its non-existent cases. There are a lot of true but vacuous statements in proofs about ∅. This confuses some programmers since many systems use type safe sets, in which the class of the entities determines a class for the container set. The math does not care about "class" and only sees sets; so that an empty set of aardvarks and an empty set of zebras in mathematics are (is?) the same set. The other confusion is that ∅ is not the database Null introduced by Codd and used in relational and other query languages in 3 valued logic. Null means unknown and many statements involving Null are undecidable (neither provably true nor provably false). The empty set is not "lack of knowledge" but certainty in the nonexistence of elements in the set. Most statements beginning "for all elements in ∅" are true, but vacuous. Most statements beginning "there exist an element in ∅" are always categorically false. It is almost impossible to construct an undecidable statement about ∅. Null and ∅ are not related. "Void" can mean "invalid" or "completely empty."

**3.33**
**end node**
<topology> node in the boundary of an edge that corresponds to the end point of that edge

**3.34**
**end point**
<geometry, topology> last point of a curve

**3.35**
**error budget**
<metric> statement of or methodology for describing the nature and magnitude of the errors which affect the results of a calculation

Note 1 to entry: In the most usual case, error budgets in this document describe metric calculations using representational geometry objects to estimate real-world metrics, such as distance and area.

**3.36**
**exponential map**
<differential geometry> function that maps tangent vectors at a point to end point of geodesic beginning at that point with an exit bearing equal to that of the vector and a length equal to that of the vector

Note 1 to entry: See first geodetic problem for an explanation of the process of calculating this mapping.

**3.37**
**exterior**
<geometry, topology> difference between the universe and the closure

Note 1 to entry: The concept of exterior is applicable to both topological object and geometric object.

**3.38**
**face**
<topology> 2-dimensional topological primitive

Note 1 to entry: The geometric realization of a face is a surface. The boundary of a face is the set of directed edges within the same topological complex that are associated with the face via the boundary relations. These can be organized as rings.

**3.39**
**first geodetic problem**
direct geodetic problem
<differential geometry, geodesy> problem that given a point on a surface and the direction and distance from that point to a second point along a geodesic, determines that second point

Note 1 to entry: This "problem" defines a mapping from the vector space at a point (each vector given by a direction and a length) to points of the Figure of Earth that satisfy the problem for that direction and distance. For example, if we fix the distance "r" and take all directions, the resultant geometry is the circle centred at the original point of radius "r". This document will make heavy use of this mapping; see exponential map and the second geodetic problem.

**3.40**
**footprint**
2D extent or projection of a 3D object on a horizontal surface

**3.41**
**function**
<mathematics, programming> rule that associates each element from a domain ("source domain," or "domain" of the function) to a unique element in another domain ("target domain," "co-domain," or "range" of the function)

**3.42**
**free function**
<mathematics, programming> function in an object-oriented programming language not associated to any object class

**3.43**
**geodesic**
**geodesic line**
<differential geometry, geodesy> curve on a surface with a zero-length tangential curvature vector

Note 1 to entry: A geodesic's curvature vector is perpendicular the surface thus has the minimum curvature of any curve restricted to the surface. This is often defined as a minimal distance curve between two points, but this does not always suffice, since some points (especially on ellipsoids and spheres) are often joined by more than one geodesic. For example, on an ellipsoid the points with (φ, λ) = (0, 0) and (0, 180) are joined by four separate geodesic [2 polar (the shorter) and 2 equatorial]. The exponential map is only guaranteed to be one-to-one for a small area (depending on where the centre is and how the surface is curved).

**3.44**
**geodesic circle**
<differential geometry, geodesy> set of points an equal distance from a given point (on the datum)

Note 1 to entry: The geodesic circles centred on a pole (either one) are the lines of constant latitude. Circles in a tangent space centred on the origin (corresponding to the point of tangency) map to geodesic circles by the exponential map on the geometric reference surface centred on the point of tangency.

**3.45**
**geometric aggregate**
collection of geometric objects that has no internal structure

Note 1 to entry: No assumptions about the spatial relationships between the elements can be made.

**3.46**
**geometric boundary**
boundary represented by a set of geometric primitives that limits the extent of a geometric object

**3.47**
**geometric complex**
set of disjoint geometric primitives where the boundary of each geometric primitive can be represented as the union of other geometric primitives of smaller dimension within the same set

Note 1 to entry: The geometric primitives in the set are disjoint in the sense that no point is interior to more than one geometric primitive. The set is closed under boundary operations, meaning that for each element in the geometric complex, there is a collection (also a geometric complex) of geometric primitives that represents the boundary of that element. Recall that the boundary of a point (the only 0D primitive object type in geometry) is empty. Thus, if the largest dimension geometric primitive is a solid (3D), the composition of the boundary operator in this definition terminates after at most three steps. It is also the case that the boundary of any object is a cycle.

**3.48**
**geometric dimension**
<geometry, topology> largest number $n$ such that each point in a set of points can be associated with a subset that has that point in its interior and is topologically isomorphic to $\mathbb{E}^n$, Euclidean $n$-space

Note 1 to entry: Curves, because they are continuous images of a portion of the real line, have geometric dimension 1. Surfaces cannot always be mapped to $R^2$ in their entirety, but around each point position, a small neighborhood can be found that resembles (under continuous functions) the interior of the unit circle in $R^2$, and are therefore 2-dimensional. In this document, most surfaces (instances of Surface) are mapped to portions of $R^2$ by their defining interpolation mechanisms.

**3.49**
**geometric object**
<geometry> spatial object representing a geometric set

Note 1 to entry: A geometric object consists of a geometric primitive, a collection of geometric primitive, or a geometric complex treated as a single entity. A geometric object may be the spatial representation of a feature object.

**3.50**
**geometric primitive**
<geometry> geometric object representing a single, connected, homogeneous (isotropic) element of space

Note 1 to entry: Geometric primitives are non-decomposed objects that present information about geometric configuration. They include points, curves, surfaces, and solids. Many geometric objects behave like primitives (supporting the same interfaces defined for geometric primitives) but are actually composites composed of some number of other primitives. General collections may be aggregates and incapable of acting like a primitive (such as the lines of a complex network, which is not connected and thus incapable of being traceable as a single line). By this definition, a geometric primitive is topological open, since the boundary points are not isotropic to the interior points. Geometry is assumed to be closed. For points, the boundary is empty.

**3.51**
**geometric realization**
<geometry, topology> geometric complex where the geometric primitives are in a 1-to-1 correspondence to the topological primitives of a topological complex, such that the boundary relations in the two complexes agree

Note 1 to entry: In such a realization, the topological primitives are considered to represent the interiors of the corresponding geometric primitives even though the primitives themselves are closed.

**3.52**
**geometric reference surface**
<geometry> surface in some Euclidean space, usually $\mathbb{E}^3$, that represents an approximation to the surface of the Earth possibly restricted to a small area but often covering the entire globe

**3.53**
**geometric set**
<geometry> set of points

Note 1 to entry: This set in most cases is infinite, except where the set consists of a list of point locations. Curves, surfaces and volumes being continuous are infinite sets of points. Some systems will define 'degenerate' curves, which are actually points.

**3.54**
**inner product**
<vector geometry> bilinear, symmetric function from pairs of vectors $\langle \vec{v}_1, \vec{v}_2 \rangle \rightarrow \mathbb{R}$ to a real number such that $\langle \vec{v}, \vec{v} \rangle = \|\vec{v}\|$ and $\langle \vec{v}_1, \vec{v}_2 \rangle = \|\vec{v}_1\| \|\vec{v}_2\| \cos(\theta)$ where "$\theta$" is the angle between $\vec{v}_1$ and $\vec{v}_2$.

Note 1 to entry: Inner products in differential geometry are used on the differentials that make up the local tangent spaces. In this document, this will usually be vectors tangent to a datum surface embedded a geocentric $\mathbb{E}^3$ Euclidean/Cartesian space.

**3.55**
**isolated node**
<topology> node not related to any edge

Note 1 to entry: Isolated nodes are usually not discussed in most algebraic or combinatorial topology. In a network, they are not interesting because they cannot be a part of any path. Algebraic topology always uses topological systems there the boundary of any object of dimension" n" is always a union of a set of objects of dimension "n–1". Which is why the dimension of the empty set is "–1".

**3.56**
**isomorphic**
<mathematics> having an isomorphism

**3.57**
**isometry**
<mathematics> mapping between metric spaces that preserves the metric

Note 1 to entry: The formal definition is $f : X \rightarrow Y \ni \forall x, y \in X . distance(x, y) = distance(f(x), f(y))$.

**3.58**
**isomorphism**
<mathematics> relationship between two domains (such as two complexes) such that there are 1-to-1, structure-preserving functions from each domain onto the other, and the composition of the two functions, in either order, is the corresponding identity function

Note 1 to entry: A geometric complex is isomorphic to a topological complex if their elements are in a 1-to-1, dimension- and boundary-preserving correspondence to one another.

$$[A, B \text{ isomorphic}] \Leftrightarrow$$
$$[\exists f : A \to B, g : B \to A \ni f \circ g = Id_A, g \circ f = Id_B] \quad (3)$$

**3.59**
**homomorphism**
<mathematics> relationship between two domains such that there is a structure-preserving function from one to the other

Note 1 to entry: A homomorphism is distinct from a isomorphism in that no inverse function is required. In an isomorphism, there are two homomorphisms that are functional inverses of one another. Continuous functions are topological homomorphisms because they preserve "topological characteristics". The mapping of topological complexes to their geometric realizations preserves the concept of boundary and is therefore a homomorphism.

**3.60**
**maximum**
**max**
**least upper bound**
**lub**
<mathematics> smallest element larger than or equal to all elements of a set contained in an ordered domain (≤)

Note 1 to entry: $[\forall a \in A \Rightarrow \max(A) \geq a] \Rightarrow [\forall b \ni [(b \ni [\forall a \in A \Rightarrow b \geq a] \Rightarrow [\max(A) \leq b]]$. Any number is an upper bound of ∅ (empty set) as a set of numbers, because any given number is greater than any number in ∅ (an admitted vacuous statement since there is no number in ∅, but true nonetheless). This means that the max (∅) must be smaller than any number; thus "−∞".

**3.61**
**metric operation**
**measure**
operations associated to measurements

Note 1 to entry: Generically, a metric is a standard of measure of any kind. The Greek word metron, meaning “measure,” giving us “metr”. Latin metricus and Greek metrikós "of, relating to measuring". See *metric unit* (3.62).

**3.62**
**metric unit**
unit of measure

Note 1 to entry: See *metric operation* (3.61).

**3.63**
**minimum**
**min**
**greatest lower bound**
**glb**
<mathematics> largest element smaller than or equal to all elements of a set contained in an ordered domain (≤)

Note 1 to entry: $[\forall a \in A \Rightarrow \min(A) \leq a] \Rightarrow [\forall b \ni [(b \ni [\forall a \in A \Rightarrow b \leq a] \Rightarrow [\min(A) \geq b]]$ Any number is a lower bound of ∅ considered as a set of numbers, because any given number is less than any number in ∅ (an admitted vacuous statement since there is no number in ∅, but true nonetheless). This means that the min(∅) must be greater than any number; thus "+∞".

**3.64**
**monotonic**
<mathematics> never increasing or never decreasing

Note 1 to entry: An increasing sequence never gets smaller. A strictly increasing sequence always gets larger. A decreasing sequence never gets larger. A strictly decreasing sequence always gets smaller.

$$
\begin{aligned}
[x(i) \text{ is increasing}] &\Leftrightarrow [i < j \Rightarrow x(i) \leq x(j)] \\
[x(i) \text{ is strictly increasing}] &\Leftrightarrow [i < j \Rightarrow x(i) < x(j)] \\
[x(i) \text{ is decreasing}] &\Leftrightarrow [i < j \Rightarrow x(i) \geq x(j)] \\
[x(i) \text{ is strictly decreasing}] &\Leftrightarrow [i < j \Rightarrow x(i) > x(j)] \\
[x(i) \text{ is decreasing}] \vee [x(i) \text{ is increasing}] &\Leftrightarrow \left[x(i) \text{ is monotonic}\right]
\end{aligned} \tag{4}
$$

**3.65**
**n-disc**
<topology, geometry> geometry isomorphic to the set of points X in $\mathbb{E}^n$ such that $\|X\| \leq 1$; set of all points in $\mathbb{E}^n$ less than or equal to one-unit distance from the origin

Note 1 to entry: The 0-disc is a point. The 1-disc is a line. The 2-disc is a circle and its interior. The 3-disc is a sphere and its interior (a ball).

**3.66**
**n-simplex**
<geometry, topology> convex hull of n+1 points in general position in a space of dimension at least n, or a topologically isomorphic image of such a geometry

Note 1 to entry: An n-simplex is n-dimensional, and is topological isomorphic to an n-disc. A 0-simplex is a point. A 1-simplex is a curve (usually a line) with two 0-simplexes (points) on its boundary; a 2-simplex is a triangular surface with three 1-simplices on its boundary. In general, an n-simplex boundary will have n (n–1)-simplexes. An n-simplex and an n-disc are topologically isomorphic. The boundary of an n-simplex is isomorphic to an (n–1)-sphere. In all cases, the number prefix represents the topological dimension of the object.

**3.67**
**n-sphere**
<geometry, topology>geometry isomorphic to the set of points X in $\mathbb{E}^{n+1}$ such that $\|X\| = 1$; set of all points in $\mathbb{E}^{n+1}$ one-unit distance from the origin

Note 1 to entry: An n-sphere is isomorphic to the boundary of an n+1 disc.

**3.68**
**neighborhood**
<topology, metric spaces> open set of points containing a specified point in its interior

**3.69**
**node**
<topology> 0-dimensional topological primitive

Note 1 to entry: The boundary of a node is the empty set.

**3.70**
**normal curvature vector**
<differential geometry, geodesy> projection of the curvature vector of the curve perpendicular to the tangent plane to the surface at the point

Note 1 to entry: The normal curvature vector of a curve is dependent only on the curve and the surface (which is a constraint on the curve). The normal curvature vector of a curve restricted to a surface is parallel to the normal vector of the surface. If the normal curvature vector is equal to the curvature vector of the curve everywhere, then the curve is a geodesic.

**3.71**
**normal**
<differential geometry, geodesy> vector perpendicular (orthogonal) to the geometric object (curve or surface) at the point

Note 1 to entry: The normal will be parallel to the "upward" normal for surfaces in $\mathbb{E}^3$ defined in the surface interface in 6.4.17.2.

**3.72**
**normal section curve**
<differential geometry, geodesy> plane curve segment containing the normal at one of its terminal points

Note 1 to entry: The usual construction begins by choosing one of the end points, the normal to the surface at that end point, and the location of the other end point and creating a plane in $\mathbb{E}^3$, which is then intersected with the datum. This curve is a normal section curve between these two points, containing the normal to the surface at the first point. On the sphere, this is the geodesic, and the two normal section curves are equal. On an ellipsoidal datum, the same is true for two points on the equator, or two points on the same meridian plane, but false in general. The reason is that in these cases the plane through the two points and the centre of the ellipsoid are planes of symmetry for the ellipsoid dividing the ellipsoid into symmetric "mirror" image halves. In general, the geodesic between the two points lies between the two normal section curves.

**3.73**
**open set**
<metric, topology, geometry> containing a metric or topologically open neighborhood of each of its points

Note 1 to entry: In a metric space, a set X us open if each point "x" in the set is contained in some small ball which is a subset of X: $[X \text{ is open}] \Leftrightarrow [x \in X] \Rightarrow [\exists \varepsilon > 0 \ni \text{distance}(x, y) < \varepsilon] \Rightarrow [y \in X]$. A topology is a set of subsets of a space which are considered open.

**3.74**
**partition of unity**
<mathematics> set of real-valued functions all over the same domain whose arithmetic sum at every domain value is 1

**3.75**
**path connected**
property of a geometric object implying that any two points on the object can be placed on a curve that remains totally within the object

Note 1 to entry: There are more general geometries that are connected but not path connected, but they are not representable as collections of "digital" geometries defined in this document. No geometric objects constructible by the methods in this document on a standard digital computer can be connected but not path connected. See the notes at the term "connected".

**3.76**
**plane curve segment**
**plane curve**
<geometry> curve in $\mathbb{E}^3$ that is contained in a plane

Note 1 to entry: In a 2-dimensional coordinate system, the test for plane curve to be valid is done in the 3-dimensional space of the geometric reference surface or coordinate reference system datum.

**3.77**
**planar topological complex**
<topology> topological complex that has a geometric realization that can be embedded in Euclidean 2 space ($\mathbb{E}^2$)

**3.78**
**Pythagorean metric**
<Euclidean geometry> distance measure on a $\mathbb{E}^n$ coordinate space using a root-mean sum of the differences between the individual coordinate offsets

Note 1 to entry: $P=(p_i), Q=(q_i), distance(P,Q)=\sqrt{\sum_{i=1}^{n}(p_i-q_i)^2}$ . The proofs of the Pythagorean metrics all depend on the local "flatness" of the space. Cartesian coordinate space which have Pythagorean metrics are called Euclidean spaces ($\mathbb{E}^n$). In the realm of coordinate reference systems, only "Engineering Coordinate Systems" are Euclidean. Any CRS using a curved Datum are by definition non-Euclidean, and cannot "truthfully" use Pythagorean metrics except for approximation. These approximations are valid for topological statements, but not for real world measures without adjustments.

**3.79**
**rhumb line**
**loxodrome**
<geometry, navigation> curve which crosses meridians of longitude at a constant bearing

**3.80**
**range**
**co-domain**
<mathematics> acceptable target values of a function

**3.81**
**row-major form**
<mathematics, computer science> storage mechanism for multidimensional array in linear memory, organized such that each row is stored in consecutive locations and such that the complete rows are the stored one after the other and continuing on is a similar fashion of each additional index

Note 1 to entry: If the indexes are (i, j) with the number of rows "r" and columns "c", then the mapping between the multidimensional locations to the linear storage locations is given by:

$$\begin{aligned}&[i,j\in\mathbb{Z}\ni 1\le i\le r;1\le j\le c]\Rightarrow[(i,j)\to(i-1)c+j]\\&[i,j,k\in\mathbb{Z}\ni 1\le i\le r;1\le j\le c;1\le k\le f]\Rightarrow[(i,j,k)\to((i-1)c+j-1)f+k]\end{aligned}\tag{5}$$

Note 2 to entry: The matrix $\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}$ in row major form is stored as [1 2 3 4 5 6 7 8 9]. For higher dimensions, the same pattern is applied recursively.

**3.82**
**second geodetic problem**
**inverse geodetic problem**
<differential geometry> problem that given two points, determines the initial direction and length of a geodesic that connects them

Note 1 to entry: See the first geodetic problem, which reverses this operation.

**3.83**
**segment**
<topology, geometry> minimal subpart of a geometry, usually as part of a composite

**3.84**
**simple**
<topology, geometry> homogeneous (all points have isomorphic neighborhoods) and with a simple boundary

Note 1 to entry: The interior is everywhere locally isomorphic to an open disc in a Euclidean coordinate space of the appropriate dimension $D^n = \{P|\|P\| < 1,0\}$. The boundary is a dimension one smaller. This essentially means that the object does not intersect nor touch itself. Generally used for a curve that does not cross not touch itself with the possible exception of boundary points. Simple closed curves are isomorphic to a circle.

**3.85**
**spatial dimension**, adj
<topology, geometry> number of independent decisions in a coordinate system required to locate a position

Note 1 to entry: This definition is logically equivalent to the topological dimension of spatial projection of the CRS. It describes the space as a target for geometry. Grammatically, the term can be a noun but used to describe space, as "the spatial dimension of CRS84 is 2".

**3.86**
**spatial dimension**, noun
<topology, geometry> any of the independent decisions made in a coordinate system to locate a position

Note 1 to entry: "A Euclidean space with a spatial dimension of 3, $E^3$, usually uses axis names 'x', 'y', and 'z', Its first spatial dimension is 'x', its second is 'y' and its third is 'z'." In the context of the space, the adjective use describes the space but the singular noun can use a name for an axis separately.

**3.87**
**spatial object**
<topology, geometry> object used for representing a spatial characteristic of a feature

**3.88**
**spatial operator**
<topology, geometry> function or procedure that has at least one spatial parameter in its domain or range

Note 1 to entry: Any UML operation on a spatial object would be classified as a spatial operator as are the query operators in Clause 8 of this document.

**3.89**
**start node**
<topology, graph theory> node in the boundary of an edge that corresponds to the start point of that edge as a curve

**3.90**
**start point**
<geometry> first point of a curve

**3.91**
**subcomplex**
<topology, geometry> complex all of whose elements are also in a larger complex

Note 1 to entry: Since the definitions of geometric complex and topological complex require only that they be closed under boundary operations, the set of any primitives of a particular dimension and below is always a subcomplex of the original, larger complex. Thus, any full planar topological complex contains an edge-node graph as a subcomplex.

**3.92**
**tangent**
<differential geometry, calculus> direction indicating the instantaneous direction of a curve at a point

Note 1 to entry: The tangent is usually calculated by differentiation of a functional representation of a curve but it may be approximated by a secant (double intersection) line from the point passing through another nearby point on the curve. The closer the second point is to the first, the better the approximation of the tangent's direction.

**3.93**
**tangential curvature vector**
**geodesic curvature vector**
<differential geometry, geodesy> projection of the curvature vector of a curve onto the tangent plane to the surface at the point

**3.94**
**tangent space**
**tangent plane**
collection of tangent vectors for curves passing through the point

**3.95**
**tangent vector**
first derivative of a curve parameterized by arc length

Note 1 to entry: If $c(s) = [(x(s), y(s), z(s)]$is a curve in a 3D Cartesian space ($\mathbb{E}^3$), and $s$ is arc length along $c$, then the tangent is $\vec{\tau}(s) = \dot{c}(s) = (\dot{x}(s), \dot{y}(s), \dot{z}(s))$, i.e. the derivative of the coordinate values of $\boldsymbol{c}$ with respect to $\boldsymbol{s}$. The curvature vector is $\kappa(s) = \ddot{c}(s) = (\ddot{x}(s), \ddot{y}(s), \ddot{z}(s))$.

**3.96**
**topological boundary**
<geometry, topology> boundary represented by a set of oriented topological primitives of smaller topological dimension that limits the extent of a topological object, or by the geometric realization of such a set

Note 1 to entry: The boundary of a topological complex is the boundary of the geometric realization of the topological complex.

**3.97**
**topological complex**
<geometry, topology> collection of topological primitives that is closed under the boundary operations

Note 1 to entry: Closed under the boundary operations means that if a topological primitive is in the topological complex, then its boundary objects are also in the topological complex.

**3.98**
**topological expression**
<geometry, topology> collection of oriented topological primitives which is operated upon like a multivariate polynomial

Note 1 to entry: Topological expressions are used for many calculations in computational topology.

**3.99**
**topological object**
<geometry, topology> spatial object representing spatial characteristics that are invariant under continuous transformations

Note 1 to entry: A topological object is a topological primitive, or a topological complex.

**3.100**
**topological primitive**
<geometry, topology> topological object that represents a single, homogeneous, non-decomposable element

Note 1 to entry: A topological primitive corresponds to the interior of a geometric primitive of the same dimension in a geometric realization.

**3.101**
**topological solid**
<geometry, topology> 3-dimensional topological primitive

Note 1 to entry: The boundary of a topological solid consists of a set of directed faces.

**3.102**
**type coercion**
<programming> conversion of one type of value to a value of a different type with similar content

Note 1 to entry: Point and DirectPosition are informationally identical (related to each other in a one to one fashion) in this context. Given a DirectPosition, a Point can be constructed. Given a Point, a DirectPosition can be derived from of its coordinates. If coercion is supported, a Point may be used where a DirectPosition is requested, and vice versa. Most programming languages use coercion, but others use "cast" operators (a type of constructor) requiring the programmer to initiate the coercion. "Strong substantiality" is related in that a subtype instance can always be coerced to any of its supertypes.

# 4 Symbols, notation and abbreviated terms

## 4.1 Presentation and notation

### 4.1.1 Unified Modeling Language (UML)

In this document, conceptual schemas are presented in the Unified Modeling Language (UML). ISO 19103 conceptual schema language presents the specific profile of UML used here. In general, UML 2 is used and most of the classifiers from ISO 19107 which use the stereotype «type» are stereotyped «interface» in this document. This document does not mandate any specific implementation system or language. Implementations do not have to use "class specification" and may use alternate implementation mechanism if they can create digital "objects" instances that meet the logical intent of the requirements stated in this document.

### 4.1.2 Naming conventions

Where possible when a classifier represents the common behavior of a set of defined things from the terms defined in Clause 3, the UML classifier will generally use the defined terms as its name. Since classifier names are capitalized and contain no space, and the defined term may contain several words, the classifier name will separate words using upper-camel-case concatenations (no spaces but each word beginning with a capital with all other letters in lowercase). Similarly, the name may be some simplified key phrase. This "UpperCamelCase" rule is generally followed but may be violated if clarity or consistency with other standards is improved by minor violations. For example;

— Point values may be represented by the datatype DirectPosition or the interface Point.

— Instances of other geometry primitives will realize the interface Primitive in the package Geometry, and other interfaces for their specific dimension and interpolation mechanism. (Curve, Surface, Solid, PolynomialCurve, Bspline, etc.

— The abstract root interface of the classes is also called Geometry.

— Instances of curves will realize the interface Curve.

— Any class name referenced from another standard will retain its original format.

Other names in the UML models may similarly use key phrases in lowerCamelCase (same as UpperCamelCase, but the first word begins in a lowercase letter). For example, the boundary, dimension, endPoint and startPoint are all used as names. Operations that create instance of the named classifier will use UpperCamelCase to parallel the classifier name that it instantiates.

## 4.2 Organization

The clauses in this document are organized in terms of UML packages. Inter-package dependencies define the criterion for viable application schemas; an application schema that contains an implementation of any package defined from this document will also contain implementations all its dependencies.

The most common keyword for a package in this document is «requirementsClass». Such packages will correspond to a conformance class in the sense that each conformance test in a conformance class will test one or more requirement referencing classifiers in the corresponding requirements class stated in this document. Keywords comparisons in UML are case-insensitive, but to remain consistent, all keywords and stereotypes are used in lowercase.

## 4.3 Abbreviated terms and symbols

The following abbreviated terms, symbols and names are important key to understanding the normative or informational content to follow. They are in general known publicly and readers of this document will need to understand their meaning. The following are useful in understanding this document:

| | |
|---|---|
| ∅ | Empty Set, the unique un-typed set containing no elements |
| ≅ | Logically equivalent |
| $( )^n$ | Implies a cross product space of "n" copies of the base, n = 0, 1, 2, ...; n=o implies a zero-length tuple; the cardinality of $\mathbb{R}^n$ for n=0 is one, the empty tuple |
| « » | Limits of a UML stereotype or keyword, either in the text or in a diagram, the symbols are called "guillemets" |
| 2D | 2-dimensional (meaning the geometric dimension of spatial portion of the CRS) |
| 3D | 3-dimensional (meaning the geometric dimension of spatial portion of the CRS) |
| API | Application Programming Interface |
| ATS | Abstract Test Suite |
| C++ | Programming language based on C with object-oriented extensions, "c plus plus" |
| CAD/CAM | Computer Aided Design, Drafting or Manufacture (usually geometry based design systems) |
| CRS | Coordinate Reference System, usually a reference to an instance of the classifiers in ISO 19111 |
| CS | Coordinate System |
| DMS | Degrees (°), minutes ('), seconds (") – representation of an angle such as 34° 32' 15" |
| $\mathbb{E}^n$ | Euclidean n-space (n dimensional Euclidean geometry space including the Pythagorean metric, usually 2 or 3 dimensional, spatial) |
| ECEF | Earth Centred, Earth-Fixed (an $\mathbb{E}^3$ space containing the ellipsoid of choice) |
| ECR | Earth Centred Rotational, alternate name for ECEF |
| JavaScript | Prototype-based scripting language with objects, dynamic typing and first-class functions |

| | |
|---|---|
| JSON | JavaScript Object Notation |
| LISP | Programming language based on LISt Processing. The standard is Common LISP. |
| MBR | Minimum Bounding Region (sometimes Rectangle) |
| OCL | Object Constraint Language (as defined by the UML standards) |
| Per | Permission |
| $\mathbb{R}$ | The real numbers |
| Rec | Recommendation |
| REQ | Requirement |
| $\mathbb{R}^n$ | Tuples of "n" Real numbers, possibly non-Euclidean, non-Cartesian |
| $\mathbb{R}^{n,1}$ | $\mathbb{R}^n$ in homogeneous coordinates $\left[(x,y)\in\mathbb{R}^2\rightarrow(wx,wy,w)\in\mathbb{R}^{2,1}\ if\ w\neq 0\right]$; although w is an ordinate the dimension of $\mathbb{R}^{n,1}$ is "n" since $\forall w:(x,y,1)=(wx,wy,w)$ |
| RSID | Reference System Identifier, or Reference System Identity |
| SQL 3 | Common name for SQL 99 during its development |
| SQL 99 | SQL language adopted in 1999, which includes object-oriented datatypes |
| SQL/MM | SQL Multi Media extensions for SQL, see ISO/IEC 13249 3 |
| TEN | Tetrahedron Irregular Network (no specific constructor is implied) |
| TIN | Triangulated Irregular Network (no specific constructor is implied) |
| UML | Unified Modeling Language |
| URI | Uniform Resource Identifier (IETF RFC 3986) |
| WKB | Well-Known Binary (data represented as standardized structured binary) |
| WKT | Well-Known Text (data represented as standardized structured text) |
| $\mathbb{Z}$ | The integers |
| $\mathbb{Z}+$ | The positive integers, often used as an index of ordinal numbers, i.e. $1^{st}$, $2^{nd}$, $3^{rd}$, ... |
| $\mathbb{Z}^n$ | Tuples of "n" integers |



# 5 Conformance

## 5.1 Requirements class conformance targets

### 5.1.1 Conformance targets

This document presents conceptual schemas for describing the spatial characteristics of geographic features. This schema defines conceptual interfaces that may be realized in application schemas, profiles and implementation specifications. The document concerns ONLY externally visible interfaces and places no restriction on the underlying implementations other than what is needed to satisfy the interface specifications. Application schema structural examples are given in Annex B.

Standardization targets (things which are testable against the Conformance Classes) determine the depth to which tests are made. This document determines tests for three classifications of target types. These are:

— Data structures for data interchange, messaging or transfers using encoding similar to those defined in ISO 19118[3]. These formats may be text or binary based. Text formats include any use or well-known text formats, or textual encodings such as XML, or JSON. Binary transfer can be associated to the in-core storage formats of any sufficiently robust programming language;

— Interfaces to software libraries and services; or

— Interfaces to databases using query languages such as SQL[12] or GeoSPARQL[40].

Each interface will have a default constructor with an associated datatype that describes one common creation operation (constructor) based on a minimal and logically complete representation of that interface's datatype implementation. Interchange and encoding will be based on these default constructors and datatypes.

Compliance to a conformance class by a "datatype" structure is a logical conformance, not a specific structure. If the datatypes implemented correspond in its informational content to the default constructors, they are deemed compliant. The following requirement defines the test more concisely.

**REQ. 1** **Conformant data structures shall be logically equivalent to the default constructor for those interfaces supported by a requirements class.**

**Rec. 1** **Equivalence should be demonstrated by a one to one bidirectional association between the values of the conformant datatype values and the possible values of the classifiers in the corresponding requirements class in this document.**

Two structures or statements are logically equivalent (≅) if each can be derived or its truth determined (⇒) from the other. For two logical statements or constructs A and B:

$$\left[A \cong B\right] \Leftrightarrow \left[\left[A \Rightarrow B\right] \wedge \left[B \Rightarrow A\right]\right] \tag{6}$$

The mechanism defined in the above recommendation essentially makes all implementations compliant to ISO 19107 also compliant as a data application to this document. For example, all arcs constructors in ISO 19107 can be used to calculate the parameters of the default arc data constructor (see 7.9.3) which is the arc centres and a set of arc endpoints.

Few applications will require the full range of capabilities described by this conceptual schema. This document, therefore, defines a set of conformance classes that will support applications whose requirements range from the minimum necessary to define data structures to full object implementation. The conformance classes are arranged as a core set of functionality with an orderly progression of extensions, so that a partial implementation of this document has a logical path to the functionality of the extensions.

Implementations that define full object functionality will allow the implementation of all operations and support logical access to the conceptual attributes and associations defined by the classifiers of the chosen conformance classes.

These implementations will normally be programming language dependent, but there are no specific language restrictions. UML is generally used for object-oriented languages with class systems, but as long as instances can be created that support the functions required, the use of a class system is an implementation decision. For example, C++ or JAVA may directly implement the UML model in this document, but JavaScript (which does not have a class-based system) is still capable of implementing object instances that can support the class contracts in the conformance classes in this document.

Implementations that choose to depend on external "free functions" for some or all operations, or forgo them altogether, will support datatypes sufficient to record the state of each of the chosen classifier as defined by its member variables and operations.



The support of sufficient data implies that all functions defined here can be implemented based sole on these data descriptions. Common encodings may include XML, Well Known Text, marshalling languages such as JSON (JavaScript Object Notation) and any other format/language needed by an application environment.

Common names for "metaphorically identical" but technically different entities are necessary to link knowledge common to different domains or applications. The UML model in this document defines abstract interfaces and data structures, application schemas define conceptual classes, various software systems define implementation classes or data structures, and the encoding standard (e.g. ISO 19118) defines entity tags. These may reference the same information content. There is no difficulty in allowing the use of the same name to represent the same information content even though at a deeper level there are significant technical differences in the digital entities being implemented. This "allows" named types defined in the UML model to be used directly under the same or similar names in application schemas (see examples in Annex B). If distinctions need to be made, package name or namespace should be supplied. For example, this document defines a Geometry interface logically equivalent to the Geometry defined in ISO 19107, and gml: AbstractGeometry defined in ISO 19136 Geography Mark up Language (GML).

Because ISO 19107 is a logical subset of this document (see Annex D), all implementations conformant to ISO 19107 that have corrected the pertinent errors in that document are conformant to some conformance class of this document. Each Requirements Class described in this document will have two associated conformance classes,

a) based on data, operations, and behaviour or

b) based on data alone.

The purpose of "a type" conformance is to identify the minimal functionality for the active interoperation of geometric data applications. The purpose of "b type" conformance is to enable universal communication of spatial information between applications consistent with this document and its predecessor ISO 19107 and thereby defining a consistent and universal information community for geometry in spatial applications. For example, all versions of GML would be in "b type" conformance.

**REQ. 2** **An “a-type” or “behavioral” conformant implementation of any requirements class in this document shall be able to satisfy all requirements in that Requirements Class.**

**REQ. 3** **A “b-type” or “data” conformant implementation of any requirements class in this document shall be able to communicate spatial information (coordinate, geometry or topology as appropriate) based on the default constructors defined in that requirements class or logically informational-equivalent data structures whose transformation to the default constructor data types by a well-known and well-defined logical mechanism.**

There are many conformant options for schemas that define types for geometric and topological objects; they are differentiated based on differences in the following criteria:

— Geometry metrics (geodesy);

— Topological dimensionality;

— Interpolation schemes;

— Structural complexity;

— Functional completeness.

### 5.1.2 Geometry metrics (geodesy)

In defining the geometry metrics, the application schema will be required to specify which underlying surfaces will be supported in calculating geometric measures. In general, the underlying surfaces (called "GeometricReferenceSurface") are:

— The 2D plane or map geometry uses standard Cartesian geometry and Pythagorean "flat" metrics, and has inherent inaccuracies associated with the curvature of the Earth dependent on the size of the area covered and the characteristics of the chosen projections (coordinate reference systems or CRS).

— A 2D sphere uses classical spherical surfaces (a perfectly round Earth of constant curvature). These systems use great circles as geodesics and do not adjust for the eccentricity of the Earth's surface. Depending on the CRS, slightly different spheres may be chosen for a "best fit" for a limited locale. While theoretically more accurate than planar, as the area of coverage increases accuracy of measurements do degrade but at a slower rate.

— A 2D ellipsoid uses a surface generated by the rotation of an ellipse about its shorter axis (polar) creating a circular equator and other lines of constant latitude, with orthogonal, elliptical lines of constant longitude. These systems adjust for the eccentricity of the ellipsoid, but do not factor in local gravitational anomalies. Again, varieties of "best fit" ellipsoids are used.

— A continuous 2D surface of an equipotential surface of gravity (such as the geoid or quasi geoid) or a tidal Reference Surface (such as mean sea level or lowest astronomic tide); Potential (or height) differences are referenced to an ellipsoid.

NOTE Although generally smooth and spatially continuous in two dimensions, such a surface is irregular and there is insufficient information to compute across it. It is usually represented through an analytical approximation using spherical harmonics where the spatial context is referenced to an ellipsoid, such as the Earth Gravity Model 2008 (EGM 2008) referenced to the WGS 84 ellipsoid, or by a digital elevation model referenced to a 2D plane.

Three dimensional geometry models will also be supported. There are essentially two approaches:

— Geocentric Cartesian 3D, using a 'flat' 3D Euclidean space: These include geocentric (origin at the centre of the Earth) and projective (origin on the surface of the Earth).

— Using one of the surfaces from the four classes above, and adding a measure of the distance in 3D either above or below the "GeometricReferenceSurface": The third dimension is measured parallel to a near normal or normal line (orthogonal to the surface, in the general direction of gravitational up).

### 5.1.3 Topological dimensionality

In defining the topological dimensionality of object types to be implemented, the application schema will be required to specify which of the interpolation types for curves or surfaces they wish to implement.

There are four levels for simple spatial geometry:

— 0-dimensional objects (points and locations in a well-defined coordinate system);

— 0-dimensional and 1-dimensional objects (adding curves of various types);

— 0-, 1-, and 2-dimensional objects (adding surfaces or areas of various types);

— 0-, 1-, 2- and 3-dimensional objects (adding solids).

### 5.1.4 Interpolation schemes

Curve implementations, for those application schemas including topological 1-dimensional objects, will always include "linear" or "geodesic" interpolation. Application schema including 1 dimensional objects should always include a mechanism to approximate any curve as a line or geodesic string to allow for transfer of data into simpler schemas when needed.



Surface implementations, for those application schemas including 2 dimensional objects, will always include a "planar" or "reference surface" interpolation technique. The most common such interpolations are boundary representations, where boundary curves lie on a single surface. These include triangular irregular networks (TIN) and polyhedral surfaces. Obviously, in 2D coordinate reference systems these polygons (with any curves as boundary) are the only surface types. "Non-flat" surfaces require 3D coordinate reference systems.

**Rec. 2** **Application schema should always include a mechanism to approximate any surface as collections of planar surface segments to allow for transfer of data into simpler schemas when needed.**

Additional curve and surface interpolation mechanisms are optional, but if implemented, they will follow the definition included in this document.

There are no restrictions on extensions of this document. The most common extension themes may include, but are not limited to the following:

— Mechanisms to improve geometric measures such a Gaussian and Riemannian metric techniques and algorithms for the improvement of accuracy or efficiency in measures for length, area and volume for geometric objects;

— Interpolation techniques that extend the point, curve, surface and solid techniques established in this document, e.g. triangulated networks, tetrahedral networks, specific spline interpolation methods. These may be targeted to specific applications;

— Indexing or tiling algorithms that improve performance or efficiencies for geometries in important geographic coordinate reference systems or datums;

— Encoding techniques for geographic geometry, such as WKT, JSON or XML;

— Algorithmic improvements to the maintenance and editing of geometry and topology structures both cellular and simplicial topology;

— Dealing with spatial temporal (4D) geometries and topologies.

### 5.1.5 Structural complexity

Four levels of structural complexity are identified:

— Geometric primitives;

— Geometric complexes;

— Topological primitives and complexes;

— Topological primitives and complexes with geometric realization.

NOTE If single definitions of each component of geometry are required, then geometric complexes are introduced into the schema. Primitives within the same geometric complex share only boundaries. If the schema requires explicit topological information, then the geometric complex is expanded to include the structure of a topological complex. The most obvious implementation would have every geometry object also implement the topological object interface. The types of object included in a complex are controlled by the dimension of that complex. What is commonly called "chain node" or "link node" topology is a 1-dimensional topological complex. What is commonly called "full topology" in a cartographic 2D environment is a 2-dimensional topological complex realized by geometric objects in a 2D spatial coordinate system. A full topology in a 3D environment would be a 3-dimension topological complex realized by geometric objects in a 3D spatial coordinate system.

The third criterion is the level of functional complexity or completeness that determines the member elements (attributes, association roles and operations) of those types that are implemented.

### 5.1.6 Functional complexity

There are three levels of functional completeness as follows:

— Data types only;

— Simple data access operations;

— Complete operations.

Subclause 10.8, Requirements Class Derived Topological Relations, of this document defines Boolean operators that may be used to investigate topological and metric relations between geometric objects.

The normative content of this document is divided into the "requirements class" packages listed in the UML model and normative Clauses. Each of these sets of requirements will correspond to a conformance class in the Abstract Test suite. Each requirements class Dependency will be realized as a conformance class extension.

## 5.2 Conformance classes

To conform to this document, an implementation will satisfy the requirements of the abstract test suite (ATS) in Annex A for a specified conformance class. In general, these conformance classes can be given by "description" parameters. These parameters include the following identifiers and their list of possible values:

Geometric Reference Surface (coordinate surface representing 0-height):

— planes;

— spheres and planes;

— ellipsoids (biaxial rotated on the polar axis), spheres and planes; or

— geosurfaces and geoids, ellipsoids, spheres and planes.

Mechanism for height:

— none (2D);

— distance from the GeometricReferenceSurface (3D or compound 2+1D).

Dimension of non-empty geometry supported (empty is "–1" dimensional):

— 0-dimension, only points or direct positions;

— 1-dimension, curves and points;

— 2-dimension, surfaces, curves and points; or

— 3-dimension, solids, surfaces, curves and points.

Support of topological structures up to dimension[2)]

— no support;

— 1-dimensional network graphs, e.g. links and nodes;

— 2-dimensional graphs, e.g. networks spanned by surfaces; or

2) Higher dimension topological complexes are acceptable, but probably restricted to cellular or simplicial topological complexes, where each element is an isomorphic copy of a disk of appropriate dimensions $\left(D_n = \{\vec{p} \in \mathbb{R}^n \mid \|p\| \leq 1\}\right)$. This will allow academic theorem from algebraic topology (specifically CW or simplicial complexes) to be applied appropriately.

— 3-dimensional, e.g. as above with surfaces spanned by solids;

— Higher dimension based on application domains, see ISO 19111, usually parametric extension of a base spatial representation of at most 3D.

Support for interpolations:

— linear including small circle along a parallel of latitude or great circle along a meridian;

— planar;

— bilinear and trilinear;

— polynomials, such as bicubic;

— splines which use the vector algebra in the domain $\left(\mathbb{R}^n\right)$ of the coordinate reference system used in the direct positions;

— rhumb line curves of constant navigational bearing, crossing all meridians at the same angle;

— and others.

Euclidean geometry in local spaces (using the tangent plane) which is then mapped to the surface supported using the mapping defined by geodesics (see exponential map in the definitions below (includes geodesics, geodesic circles and other conic sections including spirals).

This means that there will be one core conformance class containing general abstract classes but implementations for point only. Each extension adds new implementations classes, clustered into packages as needed to create a logical progression of valuable extensions.

Each implementation will have a set of "codelists" that specify support for geometry classes, coordinate systems and underlying geometric surfaces (see 6.2.1).

## 5.3 Requirements classes

Each Conformance Class corresponds to a single Requirements Class listed by clause number in the Annex A. Each such pair or conformance plus requirements corresponds to a package with stereotype «requirementsClass» in the UML model. Dependencies between these packages, see Figure 1, correspond to dependencies between the classes. For example, the core requirements class is "Coordinates" in the UML. All other requirements classes are dependent on this core. The second layer is rooted at "Geometry" containing primitive such as Point, Curve, Surface, Solid or Topology. Directly below are the extensions for each of the variations of the primitive types: curve, surface, solids and topology. "Point" does not need a separate requirements class since there is only one Point interface that is in the core "Geometry" package. For Curves, Surfaces and Solids there are more conformance extensions based on interpolation type.

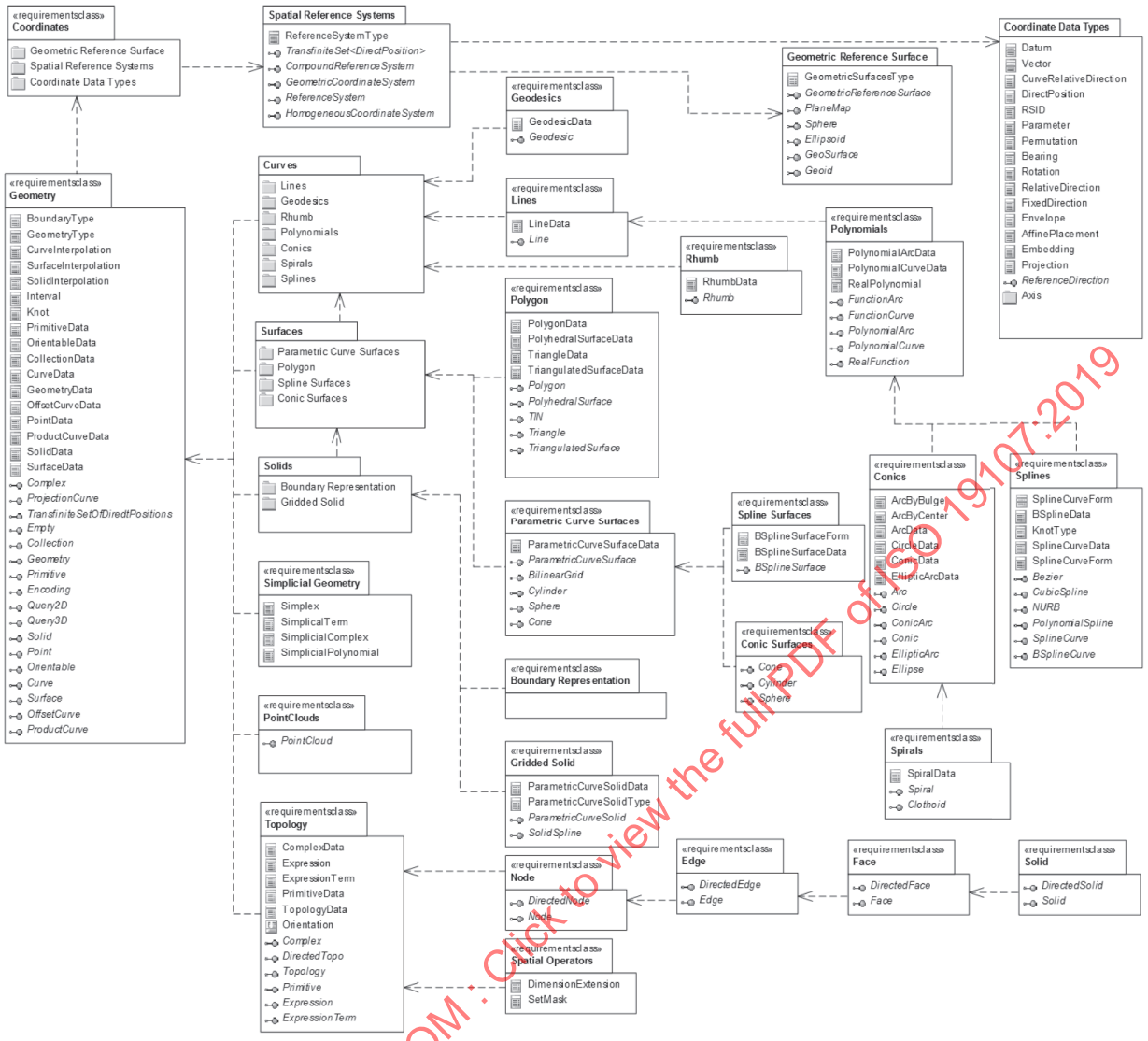



**Figure 1 — Geometry packages and requirements classes**



# 6 Coordinates and core geometry

## 6.1 Semantics

Coordinate geometry requires a coordinate space to exist. This space is usually not planar, so the space is generally not Euclidean. The first requirements class describes the geometric space and the following requirements classes describe the geometry that goes into them (Figure 1). ISO 19111 describes how an ordered list of coordinate values are used to represent a point on a geometric object called a datum. Euclidean spaces use Cartesian coordinates, $\mathbb{E}^n$, and use a Pythagorean metric. Other non-Euclidean ("curved") spaces may use $\mathbb{R}^n$ as coordinates but generally do not have Pythagorean metrics.

## 6.2 Requirements Class Coordinate

### 6.2.1 Codelists to specify capabilities

Any implementation will need to specify how deeply it has implemented the object types and other requirements. To do this a set of configurations codelists will be available as explained in 5.2[3)].

**REQ. 4** **Applications shall implement the full functionality that is specified by their claims for support of coordinate systems, coordinate reference systems, geometry types and any other parameters of conformance listed in a local codelists. The codelists that shall be available, if applicable, include GeometryType, BoundaryType, CurveInterpolation, SurfaceInterpolation, SolidInterpolation, GeometricSurfaceType, ReferenceSystemType and Datum**

NOTE Each codelist will be specified in its own separate requirement. Any such requirement is part of the testing described by REQ. 4.

### 6.2.2 Coordinate systems for Geometry — Semantics

#### 6.2.2.1 Coordinates, points and locations

The n dimensional coordinate space $\mathbb{R}^n$ consists of all n long arrays of numbers (or any type that can be interpreted as numbers). Each array of numbers represents a "point" in the "$\mathbb{R}^n$" space. In some processes or situations, this may be restricted to a subset of such points, called the 'extent of validity', and usually based on a set of constraints on values of the various offsets within the array. Each point is associated with a location, but a single location may be the target of multiple coordinate space points.

In a small region near a point in $\mathbb{R}^n$, if there is an isomorphism from this region to a geometry then the dimension of that geometry is "n" in that region. In barycentric coordinates, the mapping is restricted so that "the sum or the ordinates is 1" it restricts the coordinates to a hyperplane in $\mathbb{R}^n$ and therefore the dimension is "n–1". In homogeneous coordinates, variations only in the coordinate "w" will create a line whose coordinate sets map to the same point. Locations are given by direct positions; see 6.2.9, but the dimension of the coordinate space depends on how those coordinates are used.

EXAMPLE A 1-dimensional coordinate space may be mapped to a circle by angular displacement (in radians) in a chosen direction from a chosen origin on the circle. In this case, "x" and "x + 2π" are distinct points in the coordinate space, but represent the same angular location on the circle. Any interval of x shorter than 2π is one-to-one and invertible, so the geometric dimension is preserved. In this example,

location (DirectPosition(x)) = location (DirectPosition (x + 2π))

A coordinate space and a function that maps coordinate tuples to locations define a coordinate reference system (see ISO 19111) that is associated with an implementation of the datatype DirectPosition that uses representations from this coordinate system.



3) This implies conformance testing suites should test these functionality according to the claims of the applications documentation claim.

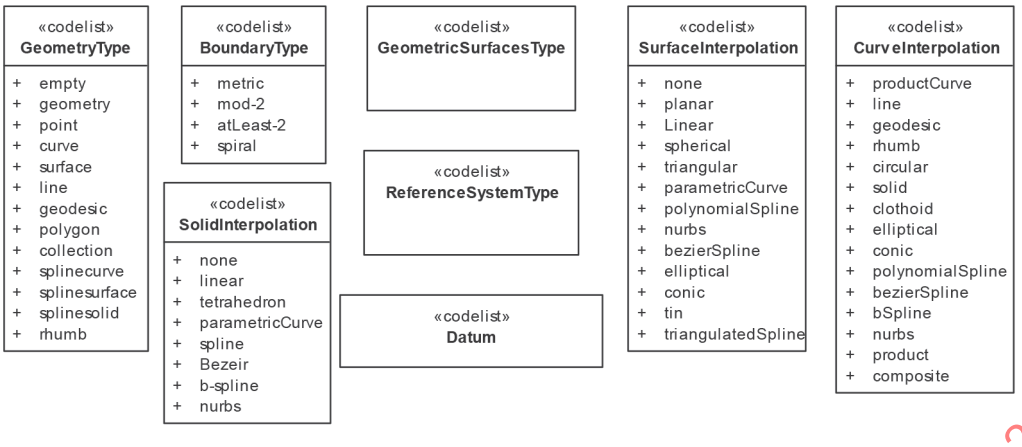



**Figure 2 — Codelists used in REQ. 4**

All geometric objects in this document are represented as mathematical or geometric constructions within a coordinate system that represents some set, usually infinite, of coordinated points, each point describable by a DirectPosition. **This document deals with both Euclidean ("flat" spaces) and non-Euclidean geometries ("curved" spaces).** A simple example of such geometry constructions would be a (coordinate) line defined by two coordinate points, and a linear interpolation in $\mathbb{R}^n$, here represented as vector $\overrightarrow{P_1}$ and $\overrightarrow{P_2}$ that represents all points in. $\left\{\vec{X} = u\overrightarrow{P_1} + (1-u)\overrightarrow{P_2} \middle| 0 \le u \le 1\right\}; \vec{X}(0) = \overrightarrow{P_1}; \vec{X}(1) = \overrightarrow{P_2}$.

NOTE This "line" in the coordinate system does not necessarily represent a line in the real, physical world, i.e. a "line of sight" projected back down onto the surface of the Earth, which is a segment of a geodesic curve path of shortest length between any two of its points which are the "great circles" on a sphere.

EXAMPLE A Mercator projection "line" is a line of constant bearing (rhumb line or loxodrome), and is either the arc of a circle (for east-west lines) or a segment of polar spiral. The only rhumb lines in a Mercator map that are geodesics on a sphere or ellipsoid are the equator or a line of constant longitude ("meridians") running north-south.

**REQ. 5** **An implementation of the Package Coordinate shall have all instances and properties specified for this package, its contents, and its dependencies, contained in the UML model for this package in this document.**

REQ 5, put simply, requires that this package needs to be implemented completely including all of its dependent packages, in particular those that may be associated to ISO 19111.

### 6.2.2.2 Coordinate systems

Many of these spaces will be coordinate reference systems as defined by ISO 19111, but not all. Nonetheless, all of these systems will satisfy some basic properties.

**REQ. 6** **Each coordinate offset in the tuples of a direct position shall contain instances of a class that can be represented as and therefore coerced into a single real number.**

**REQ. 7** **Spatial coordinates shall be consistent with a coordinate reference system defined by the mechanisms in ISO 19111.**

**REQ. 8** **Temporal coordinates shall be consistent with a reference system consistent with ISO 19108.**

**REQ. 9** **Parametric coordinate shall be consistent with the extension mechanism defined in ISO 19111.**

This does not mean that the coordinate offset will be a real number, but that it can be converted to one. For example, 4° 30' (four degrees, thirty minutes) is equivalent to 4,5° and could therefore be used as a coordinate offset. In the discussions in this document, we will assume that all coordinates have been represented by real numbers, but this does not restrict representations such as DMS (degree, minutes and seconds) as they can always be cast as a real number.

A coordinate reference system that speaks of longitude will often restrict them to the equivalent of the interval [–180°, 180°]. This causes some problems for geometries that cross the 180° meridian. The easiest work around for this sort of problem is to eliminate the discontinuity by allowing –179° and 181° both to be valid and to represent the same longitude as far as position on the earth is concerned.

**REQ. 10** **For a coordinate offset that can be extended using the period of a function, the additions or subtractions of that period shall not affect the validity and position of the DirectPosition.**

The coordinate offsets are not assumed to represent the same or even compatible units of measure. For example, in a 3D latitude longitude system, the first two offsets are angles such as in decimal degrees, and the third (usually height or, more properly, "ellipsoidal height") will be a linear unit of measure such as metres. With this and the curvature of the ellipsoid, the standard Euclidean distance formulae using the Pythagorean Theorem will not represent a distance, but will define the same set of continuous functions or topology. In other words, the underlying space is a "manifold" and is locally a continuous image of the Cartesian coordinate space of the same dimension. The continuous function will not normally be an isometry (a "preserver of distance", see the definition in Clause 3).

Any function that depends on this sort of measure, such as distance, curve length or surface area, can have two interpretations: one using the local Euclidean metrics, which are essentially unit less, and another that has been converted to an approximation of the real-world measure. This is an application option, but should be consistent with the conceptual model of the application. For example, the Earth is not a sphere, but spherical trigonometry makes spheres easier to handle than ellipsoids, and if the sphere uses a local mean radius of the Earth, the differences in length measures may exhibit only small inaccuracy, which may be sufficient for many applications.

**REQ. 11** **Metric (measurement) operations in geometry-related interfaces shall use the best approximation of real-world values practicable. A conformant application shall supply error budget information to allow the user to determine the fitness for use of the values returned.**

Almost all modern coordinate systems used in geography can be mapped back to an Earth centred; Earth fixed (ECEF) "geocentric" Cartesian coordinate system which is essentially a Euclidean space where the Pythagorean metric is valid. Mapping back to this coordinate system will always allow a final option for calculations using 3D calculus techniques. There are methods for ellipsoids (see [48] and [47]) which can calculate directly in elliptical or spherical coordinates (latitude, longitude). Less extreme, less accurate but equally valid is to map geometry to a local engineering system (map distances) where the same "Euclidean geometry" based calculations (either corrected or not) are feasible and have controllable error budgets especially in small areas.

**REQ. 12** **All locations in a list or array shall use the same coordinate system and shall reference reality in a manner representable by continuous functions from the coordinate tuples (direct positions) to reality in such a manner that "nearby" coordinates in the direct positions map to "nearby" positions in reality.**

NOTE The formal definition using topological terms of a continuous mapping:

Let $X, Y$ be topological spaces and $f : X \to Y$ is a mapping from $X$ to $Y$.

$$\left[f \text{ is continuous}\right] \Leftrightarrow \left[\forall B \subseteq Y, \left[B \text{ is open}\right] \Rightarrow \left[f^{-1}(B) \text{ is open}\right]\right]$$

This document does not assume that these functions maintain coordinate dimension. For example, direct position may be given in "homogeneous coordinates" which equates, for every w≠0, the



coordinate (*wx, wy, wz, w*) in four dimensional space $\mathbb{R}^4$ with the point (*x, y, z*) in a three dimensional space $\mathbb{R}^3$. The topology of both systems is three dimensional, since the value of "*w*" does not affect the spatial position.

Some offsets in the coordinate array of a direct position may have no spatial interpretation, such as a temporal parameter, or one that corresponds to some other measurement or quantity associated with the position being described, such as a measurement in a linear reference system, or a physical property of space, such as temperature or pressure. Such offsets will not usually affect distance measures, unless those measures are interpreted by the application in some manner.

All direct positions will have a projection into a purely spatial coordinate as would be associated with a coordinate reference system (SC_CRS) from ISO 19111. This projection is not assumed one to one, even if the coordinate system is purely spatial. It is assumed locally one to one, and bicontinuous. This means that the coordinate system will locally be reversible, but that it may wrap around singularities in the definition of the SC_CRS, such as occurs at 180° longitude line in a latitude longitude system or at the poles in such a system.

#### 6.2.2.3 Metrics and distance measures

From the above description, the "flat" concept of distance given by the Pythagorean Theorem is not directly applicable to DirectPositions except in limited circumstances.

The general solution is the use of Riemannian or Differential geometry, but this level of generality is not normally needed in geographic information. The key to the simplification is in ISO 19111, in that the datum surfaces that approximate the Earth are usually embedded in a 3D geocentric Euclidean space. This means that the geometries on this datum surface are also in $\mathbb{E}^3$. Most calculations on ellipsoids can use Vincenty's formulae[48][47][46], or in the ECEF $\mathbb{E}^3$ that contains the datum surface. The distance formulae for great circles on a sphere for two points is (angles in radians) is:

$$\begin{aligned} &\text{points} = \{(\phi_1, \lambda_1), (\phi_2, \lambda_2)\},\ r = (\text{local}) \text{ radius of the Earth} \\ &d(p_1, p_2) = r(\arccos(\sin\phi_1 \sin\phi_2 + \cos\phi_1 \cos\phi_2 \sin(|\lambda_1 - \lambda_2|) \end{aligned} \tag{7}$$

A recursive form of Vincenty's formula[47] is straight forward and it creates accurate ellipsoidal results. Several open source versions of the algorithms may be found on the web.

#### 6.2.2.4 Underlying geometric space

Geometry systems inherently make assumptions about the "space" in which the geometric objects are defined. Classical Euclidean geometry assumes that the objects lie in an infinite flat plane represented, for example, by a Cartesian coordinate space with a standard Pythagorean metric. Geographic information if restricted to relatively small areas can be analysed using classical Euclidean methods or algebraic calculations in a Cartesian coordinate space that depend on the 'flat' Pythagorean metric. For larger areas, the effects of the Earth's curvature become significant even for primitive surveying systems.

EXAMPLE Much of the United States is surveyed using the Land Ordinance of 1785, which defines Public Land Survey System (PLSS) commonly called "the Jeffersonian Grid". In this system, major boundaries are defined by "equally spaced" horizontal (east west) lines of constant latitude and vertical (north south) lines of constant longitude. Since these lines are generally the boundary lines between owned parcels, roads often follow them. Since the lines are spaced by distance (6 or 24 miles, approximately 10 or 38 kilometres), they misalign due to the curvature of the Earth, the north south roads often "jog" (meaning a brief abrupt double change in direction, "left then quick right" or vice versa) when they cross the major east west lines also being used as rights of way for other roads. The further away from the origin of the "grid" the greater the width of the "jog". Similarly, on any UTM projection, the metres easting and northing for a grid square would not be sufficient to calculate the area of that square with the distortion of the area dependent on the square's position.

In other words, Euclidean based intuition is inaccurate on a curved, and therefore non-Euclidean, surface. To make such calculations more accurate, a more accurate model of the Earth's surface is needed. The



most common models involve 2D surfaces embedded in a 3D space. The 3D space is a Euclidean space, with a Pythagorean metric represented by Cartesian coordinates. The geometric/metric structure of the surfaces is inherited from the embedding 3D Cartesian space. On the surface, the distance between points is defined by curves of shortest distance lying on the surface that joins the points in question, similarly to the way lines in flat Euclidean spaces define distances in the Pythagorean metric. Thus, the distance between points on a sphere is defined by the length of the great circle on the sphere joining the two points. For ellipsoids, the geodesics are more complex, but they are defined as curves of shortest distances between points, and their length defines the metric on the underlying surface.

The surfaces in most general use are:

— Maps – a relatively local map projection onto a 2D plane where Euclidean geometry properly scaled is a close approximation to reality,

— Spheres – a 2-dimensional sphere with a fixed radius, embeddable in $\mathbb{R}^3$, allowing the use of classical spherical geometry and trigonometry,

— Ellipsoids – a bi radial surface of revolution using an ellipse of revolution whose minor axis is approximately aligned with the Earth's polar axis, or

— Calculated GeometricReferenceSurface or Geoids – an adjustment of an ellipsoid to an equipotential surface of gravity to adjust for the variations in Earth's local density and thus it gravity potential.

In ISO 19111 a sphere can be considered to be a special case of an ellipsoid and no distinction need be made between them for CRS definitions. In this document a distinction is made because geometric calculations on a sphere use simpler formulas based on the symmetries of a sphere, e.g. those based on "great circles" geodesics, whereas many geometric calculations on an ellipsoid are more complex and may require the use of more complex approximations. An approximation is not necessarily less accurate than a closed form formula, which will use polynomial approximation for trigonometric functions, but different implementations may treat these differently.

Issues of local usage can adjust a CRS to work based on a local best fit of the Reference Surface. For example, in the United States local mapping agencies often use State Plane Coordinate system specifically to preserve the accuracy of Euclidean geometry. To do this, there are 124 distinct state plane systems used to cover the country. Such a system could use the "Reference Surface — Plane" parameter for the coordinate conformance class.



### 6.2.3 GeometricReferenceSurface

#### 6.2.3.1 Interface GeometricReferenceSurface

A GeometricReferenceSurface is any surface usually embedded in a Euclidean space used to describe a geometric space for determining various measures (e. g. length or area) (Figure 3).

The interface GeometricReferenceSurface has three attributes:

— id: URI — each GeometricReferenceSurface has a unique identity. To support Web access, the identity is a URI. In many cases this will correspond to a surface used as a datum;

— metadata: URI[1..*] — each geometric surface should have a complete set of metadata;

— datum: Datum — the actual surface (derived from metadata).

**REQ. 13** **A GeometricReferenceSurface implementation shall have all properties for that interface specified for it in the UML model in this document.**

**REQ. 14** **If the GeometricReferenceSurface is not embedded in a Euclidean space, then it shall have either a Riemannian metric or some other mechanism to determine distance and area metrics within a documented level of accuracy.**

**REQ. 15** **The use of approximations for distance and area measures shall be part of the metadata of any application**

Some ellipsoid approximations based on approximating spheres are given in Formula (7). These three items, the CRS, the GeometricReferenceSurface and the association between coordinates in the CRS and position relative to that GeometricReferenceSurface are sufficient to define how geometric calculations such as length, distance and area may be done using the geometry objects defined in this document.

**REQ. 16** **If the CRS is supplied but the GeometricReferenceSurface is unspecified, the default surface shall be the surface of the CRS object's related Datum as defined in ISO 19111.**

**REQ. 17** **When using a 2D CRS, all geometric measures, such as distance, bearing, length, area or volume, shall be consistent with the underlying GeometricReferenceSurface or Datum**

NOTE For surfaces that are more complex (ellipsoids and geoids) numeric approximations for distances and geodesic are often used. Implementations will be required to access these approximations to determine their error budget. For ellipsoids, Vincenty's formulae[47][38] may suffice.

**REQ. 18** **For each 3D CRS, the horizontal 2D measures shall be consistent with the geometric reference system, and the vertical adjustment shall be consistent with an isometric embedding of the GeometricReferenceSurface in 3D Euclidean space.**

**REQ. 19** **Each implementation system shall specify each CRS supported in a codelist.**

In a 2D system, volume measures may be calculated from footprints (horizontal extent) and separate attribute information such as average height.

#### 6.2.3.2 CodeList: GeometricSurfaceType

The codelist GeometricSurfaceType is a list of all GeometricReferenceSurface types by identify supported by the local application.

**REQ. 20** **The local instance of the codelist GeometricSurfaceTypes shall list all GeometricReferenceSurface or Datum types supported by the local application.**

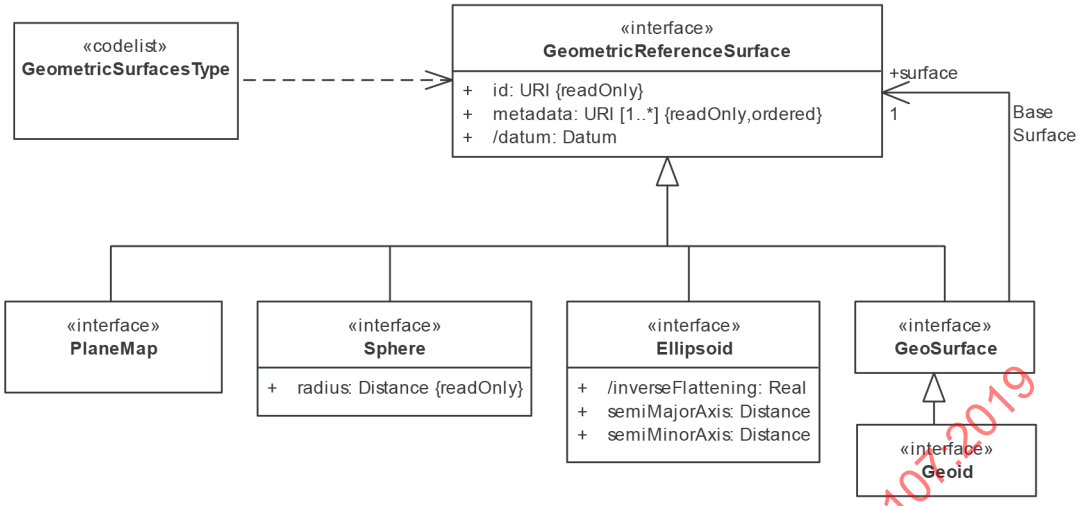



**Figure 3 — GeometricReferenceSurface**



### 6.2.3.3 Interface PlaneMap

Some geographic processing is still done using a "map metaphor" which essentially assumes that the flat image of an underlying graphic (paper or screen) differs from reality only by the scale of the map. While the versions of this assumption are varied, two cases seem to be more common than others:

1. “flat earth” assumptions, and
2. Cartographic map projections.

In the "flat earth" assumption, the plane is mapped to the Earth by assuming a single point (the "origin") of attachment (possibly given in a geographic coordinate system) and all other points defined by measured offsets and directions (vectors) from this origin. This essentially equivalent to the “exponential map” (see the definition in 3.36). This is especially useful in small areas where the curvature of the Earth makes only small variations in the measures, in general areas of less than 10 kilometres (or 6 miles). In ISO 19111, these systems are called Engineering CRSs; and may be architectural or engineering drawings using large scales.

If the map is created by using local surveying procedures, then one way to think of this may be to consider the map to be representation of the tangent plane at the origin, and the mapping to points on the map to be defined by the differential geometry "exponential map" from this tangent plane to any more accurate geometric reference system. The exponential map looks at the tangent plane in polar coordinates, being a bearing and a distance from the origin. This "polar" representation is then used to generate the geodesic on the Earth with this initial bearing, and this length and using its endPoint as the position on the Earth that is represented by the position on the map defined by this angle and this distance.

If the map plane is viewed as the x y, or $\mathbb{E}^2$, the normal to the plane is a vector representing "up". The geodesics in the plane are the straight lines, but these can only be used with any accuracy in small areas since when they are brought back to the curved Earth, their geometry changes. This essentially means that these entities use "map measures" and the CRS system in use is a projection.

Different projections may preserve different aspects of the real "curved" geometry. Some may preserve shape, angle, area or distance better than others. But, projection of a spherical or ellipsoidal model of the Earth to a plane cannot be done without some distortion. Conventional topographic maps usually preserve shape in a small area (they are often conformal maps, preserving angles, see the definition in 3.10). Mapping designed for statistical purposes may preserve area. For accurate measures of different types, different map projections may be required.

#### 6.2.3.4 Interface Sphere

The spherical approximation to the Earth's surface is one of the oldest in continuous use, and "central angle" latitude and longitude (valid for spheres but not ellipsoids) is the most common mental model of terrestrial locations (Figure 3). Further, spherical trigonometry (19th century) gives closed form solutions to most calculations needed for model-based measures.

Each sphere's shape is completely defined by its single attribute:

— radius: Distance — radius (mean radius of Earth is about 6,371,009 m)

Assuming the sphere in $\mathbb{E}^3$ is at the origin $(x, y, z) = (0,0,0)$, then $x^2 + y^2 + z^2 = r^2$. The normal vectors to the surface are the extensions of the radial lines from the centre of the sphere. The geodesics on the sphere are the great circles defined by planes through the origin and creating circular curves centred there with their normal equal to the normal of the sphere.

NOTE This means in these spherical CRS systems, 2D geodesics are great circles. The use of 3D adds offsets from the sphere and distances calculable by integrals in the embedding Euclidean space.

Math The planes that pass through the sphere's centre cut the sphere in great circles. Since these circles are contained within these planes, their curvature vectors (2nd derivatives of the curves with respect to arc length) are vectors following the radial lines of these circles, and thus perpendicular to the sphere. Therefore, the great circles have zero tangential curvature vectors and are thus geodesics by the definition in Clause 4.

#### 6.2.3.5 Interface Ellipsoid

Mathematically an ellipsoid can have 3 separate radii, but a geographic ellipsoid is defined by a surface of rotation for an ellipse, around its shorter axis (which becomes the polar axis). So, each geographic ellipsoid's shape is defined by two radii (Figure 3) for it axes:

— semiMajorAxis: Distance — largest radius, anywhere along the equator ($a$);

— semiMinorAxis: Distance — smallest radius, at the two poles ($b$).

A third alternate value, "inverse flattening", is often given, eliminating the need for the semi-minor axis.

— inverseFlattening: Real — derived value equal to $\left(\frac{a}{a-b}\right)$ which is close to 300,0.

**REQ. 21** **A compliant implementation that supports Ellipsoids as a GeometricReferenceSurfaces for their coordinate reference shall implement all metrics consistent with this ellipsoid's embedding in $\mathbb{E}^3$.**

**REQ. 22** **Distance on ellipsoids shall be determined by geodesics on the ellipsoid that are curves with a zero tangential curvature vector, or by some reasonable approximation (see REQ. 30).**

#### 6.2.3.6 Interface GeoSurface

For the purposes of this document, a geosurface is any surface defined by offset mechanism from another standard GeometricReferenceSurface (its "Base Surface"), the most common example being an isosurface of the gravity potential (Figure 3).

**REQ. 23** **Compliant implementations using a GeoSurface shall take into account the offset mechanism in calculating metric values.**

#### 6.2.3.7 Interface Geoid

For the purposes of this document, a geoid is a geosurface defined by an isosurface of the gravity potential usually using an ellipsoid as it Base Surface (Figure 3).

### 6.2.4 Interface ReferenceSystem

#### 6.2.4.1 Semantics

A reference system is usually an implementation of the datatype RS_ReferenceSystem from ISO 19111. The major difference is that the local version uses a URI as an identity, since most useful geographic information has to be in a commonly understood coordinate system meaning its and details needs to be easily available to all (Figure 4).

#### 6.2.4.2 Attribute id

**REQ. 24** **Each ReferenceSystem shall have a URI for identification.**

URI used for this purpose should be HTTP-URIs.

```
id: URI
```

#### 6.2.4.3 Attribute rsid

In many applications, such as linear reference systems (ISO 19148) and moving features (ISO 19141), extra columns are added to the coordinates for "other" non-ISO 19111 purposes. The most common practice is to concatenate reference systems, represented by listing an RSID for each component system.

```
rsid: RSID [1..*]
```

**REQ. 25** **Each ReferenceSystem object shall have a sequence of RSIDs to identify projected reference systems from which it is composed.**

**Rec. 3** **The RSID should follow a general pattern of [spatial axes] ([temporal axes]) ([homogeneous Weight]) ([parameter axes])**

#### 6.2.4.4 ttribute dimension

The attribute dimension gives the number of offsets in the coordinate array that is the same as the number of axes in the system. The ordinates and the axes are in the same order, each ordinate offset corresponding to the same offset in the axis array.

```
dimension: Integer
```

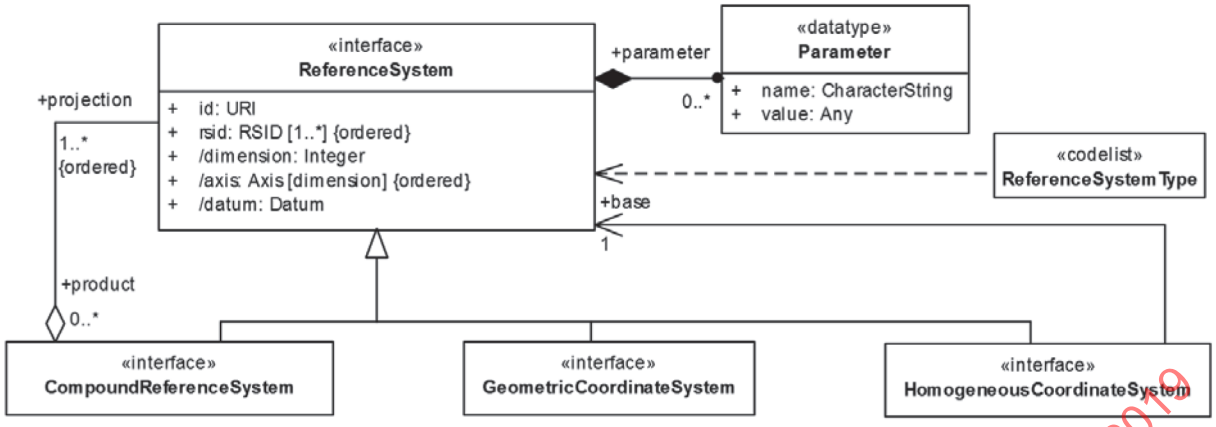



**Figure 4 — Reference systems**

#### 6.2.4.5 Attribute axis

The array axis lists the various axes from the identified reference systems (in the rsid array). Axis groups from each RSID are grouped in the order given in the corresponding reference system.

```
axis: Axis [1..*]
```

#### 6.2.4.6 Attribute datum

The attribute datum is the datum used for the spatial components of the geometry in this system.

```
datum: Datum
```

#### 6.2.4.7 Role parameter

A ReferenceSystem can have direct connections to any number of parameters (name, value pairs) that may participate in the use of the system. Most parameters may be derived from the reference system IDs (RSID).

```
parameter: Parameter [0..*]
```

### 6.2.5 Codelist ReferenceSystemTypes

The codelist ReferenceSystemTypes is a list of all primitive systems supported by the local application

**REQ. 26** **The local codelist shall list all primitive coordinate reference systems supported by the local application**

### 6.2.6 Interface CompoundReferenceSystem

#### 6.2.6.1 Semantics

A compound reference system allows the implementer of geometry entities to concatenate coordinate tuples into single coordinate direct position values. Within a single set, all spatial coordinates should be from the same coordinate reference system.

**REQ. 27** **A Reference System with more than 1 projection shall support the interface for a CompoundReferenceSystem which has an attribute projection consisting of an array of RSID for each projection.**

EXAMPLE If a reference system consists of a 2D CRS to establish horizontal positions, a 1D vertical reference system for height and a 1D temporal reference system for time, then the rsid will consist of an RSID identifier for the horizontal CRS, a second RSID identifier for the vertical system, and a third RSID for a temporal system.

A compound coordinate system is one which is composed of a sequence (represented by an ordered association role) of other coordinate systems. The dimension of a compound system is the sum of the dimensions of its projections. The spatial dimension of the space is purely dependent on the projections that are to geographic positions in the real world.

This is a logical generalization of the CompoundCRS defined in ISO 19111, which is normally restricted to adding spatial or temporal dimensions. This interface does not restrict any additional columns, and may involve dependencies between coordinate offsets.

#### 6.2.6.2 Role projection

The ordered association role "projection" of CompoundReferenceSystem lists the components of this instance of the interface, in the order that the coordinates are used in the compound reference system.

```
CompoundReferenceSystem::projection:ReferenceSystem[1..*]
```

#### 6.2.6.3 Attribute rsid

The RSID identifiers of the ReferenceSystem instances in the projection role are repeated as an array of RSID system identifiers in the inherited attribute "rsid".

```
rsid: RSID [1..*]
```

NOTE The only reordering of axis or ordinates that is usable is in the essentially local-only GeometricCoordinateSystem that can rearrange the ordinates using its local attribute permutation.

### 6.2.7 Interface HomogeneousCoordinateSystem

Homogeneous coordinates are useful in defining rational b-splines and are used in projective geometry.

**REQ. 28** **A HomogeneousCoordinateSystem shall be a GeometricCoordinateSystem that shall have an extra axis for a non-zero weight "w". All other affected offsets shall be multiplied by this "w".**

The usual mapping between the 2D base coordinate system to the corresponding homogeneous system is (x, y) → (x, y, 1) → (wx, wy, w), w ≠ 0 or in 3D (x, y, z) → (x, y, z, 1) → (wx, wy, wz, w), w ≠ 0, and so the usual projection back to the base system requires division by w as in (X, Y, W) → (X/W, Y/W, 1) → (X/W, Y/W) and so W must be non-zero. Since all the "classes" in this document are "«interfaces»", the internal representation of the coordinates is an implementation decision.

### 6.2.8 Interface GeometricCoordinateSystem

#### 6.2.8.1 Semantics

The final step in creating a coordinate system for geometric calculation suitable for geographic measures is to combine the concepts in ISO 19111 in the CRS definitions, with the calculation mechanisms described above into a single interface (Figure 5).

**REQ. 29** **Every geometry used to represent a locus of positions in a geographic space shall be associated with a coordinate reference system (CRS) as defined in ISO 19111 or one of its extensions, and to a GeometricReferenceSurface that shall be used in all "metric" calculations implemented by its object representations.**

**REQ. 30** **If a geometry instance is associated with a GeometricReferenceSurface different from the datum of the associated CRS, then the metadata associated with that geometry shall make explicit reference to this inconsistency and shall describe an an error budget for errors introduced by this use of non-standard geometric reference system.**

Figure 5 describes the object model for these basic classes dealing with the relationship of geometry to its coordinate space. This is an information model, and any implementation of this model that contains the needed information regardless of the form will be compliant.

#### 6.2.8.2 Attribute name

The attribute "name" is a human-readable name in the local language that is a mnemonic for the coordinate system.

```
GeometricCoordinateSystem::name:CharacterString
```

**Rec. 4** **Information communities should standardize the Geometric Coordinate System names for common usage and provide or specify metadata locations for all such systems.**

#### 6.2.8.3 Attribute spatialDimension

The attribute "spatial dimension" indicates the dimension of the spatial coordinate systems in use. Usual values for the spatialDimension are two and three. For spatial systems, the first two coordinates are usually "horizontal offsets". Examples include (latitude, longitude) in degrees and (easting, northing) in metres. The third dimension, at each point, is usually orthogonal, at least nominally, to the gravity isosurfaces; i.e. the usual 3rd coordinate is some form of height, often but not always in metres.

```
spatialDimension:Integer
```

**REQ. 31** **Each spatial dimension shall be represented in or transformable into measures of length or distance.**

NOTE The spatial dimension of a coordinate space is the number of decisions made to locate a point in space. The spatial dimensions in a coordinate system are those decisions. So, in a (latitude, longitude) system, the spatial dimension is two, and the spatial dimensions or axes are latitude and longitude. See definitions 3.85 and 3.86.

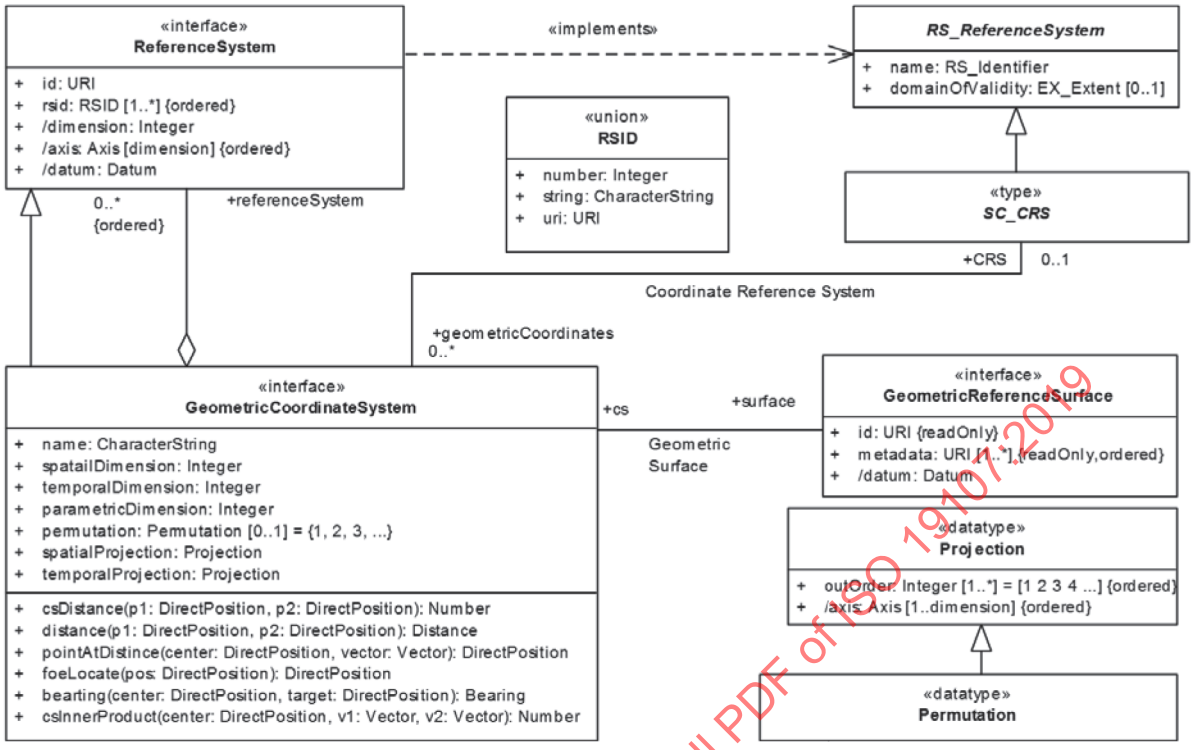



**Figure 5 — Reference System and CRS**

### 6.2.8.4 Attribute temporalDimension

The attribute "temporal dimension" indicates the number of temporal axes in the coordinate system. In a standard spatiotemporal reference system, the temporal dimension is 1 and represents either an absolute position (implying both time and date) or the time passed since an event or time implied by context. The temporal dimension may be larger, and the interpretation of each offset should be well documented and defined in the metadata for the geometric coordinate system.

```
GeometricCoordinateSystem::temporalDimension:Integer
```

### 6.2.8.5 Attribute parametricDimension

The attribute "parametric dimension" is the count of the number of undifferentiated parameters stored in the geometric coordinate system. Other than being numeric in nature, there is no a priori restriction on parameters and their interpretations. Each offset should be well documented in the metadata for the geometric coordinate system.

```
parametricDimension:Integer
```

### 6.2.8.6 Attribute permutation

The attribute "permutation" allows a local reordering of the coordinate offsets from the default one for the use of the local application. Permutations are represented by "0-up" ordinate offsets of the output. An empty permutation implies that the coordinates are not rearranged.

```
permutation:Permutation[0..1]
```

EXAMPLE A normally (latitude, longitude, height) system which is left-handed when subject to a [2, 1, 3] permutation becomes a (longitude, latitude, height) system, which is right-handed.

NOTE The most common purpose for use of an internal permutation is to maintain a "right-handed" coordinate system to allow standard mathematical calculation. For example, using Green's theorem to determine the orientation of a cycle "*c(t)*":

$$\left[\oint_{c(t)} xdy - ydx > 0\right] \Rightarrow \left[c(t) \text{ is counter} - \text{clockwise}\right] \tag{8}$$

is valid only if the (x, y) coordinate system is right-handed. If the system is left-handed, the sign of the integral is changed.

#### 6.2.8.7 Attribute spatialProjection

The attribute "spatialProjection" is a projection to a purely spatial coordinate system. Projections are represented by ordinate offsets indexed by the positive integers, $\mathbb{Z}^+ = \{1, 2, 3, \ldots\}$, of the axes in the range of the projection function.

```
GeometricCoordinateSystem::spatialProjection:Projection
```

#### 6.2.8.8 Attribute temporalProjection

The attribute "temporalProjection" is a projection to a sequence of pure temporal coordinate system from ISO 19108. The semantics of multiple temporal offsets in a reference system should be covered in the geometric coordinate system metadata.

```
GeometricCoordinateSystem::temporalProjection:Projection
```

#### 6.2.8.9 Operation: csDistance

The coordinate system distance function "csDistance" is the normal distance function for the coordinates, using the appropriate Euclidean metric function (usually the Pythagorean formulae for a Cartesian system). This metric will normally produce the same topological structure on the geometric reference surface as the "distance" function. See 3.62 metric unit.

```
csDistance(p1:DirectPosition,p2:DirectPosition):Number
```

**Rec. 5 The result of csDistance should normally correspond to a metric unit.**

#### 6.2.8.10 Operation: distance

The distance is the "real world" equivalent of the distance on the geometric reference surface, which means that this is the best approximation supplied by the application to the distance that could actually be measured between the real world positions represented by the points (see REQ. 11).

```
distance(p1:DirectPosition,p2:DirectPosition):Distance
```

In a 2D system, the distance between two points is the length of the "shortest" geodesic on the geometric reference surface joining them.

#### 6.2.8.11 Operation: pointAtDistance

The operation "point at a distance" solves the first geodetic problem that is given a starting point, an initial direction (bearing) and a distance; find a new point at that distance from the original point with that bearing as the initial tangent to the geodesic joining them. In the protocol for the operation, the parameter vector gives both the bearing (the direction of the vector) and the distance (the length of the vector).



```
pointAtDistance(centre:DirectPosition,vector:Vector):DirectPosition
```

NOTE 1 The vector space at a point is a Euclidean space of the same dimension of the underlying manifold. For a surface that is isomorphic to $\mathbb{E}^2$.

Semantically, the centre point and the vector are an element of the tangent space on the GeometricReferenceSurface at the centre point. This operation is called the exponential map for the GeometricReferenceSurface at the point, mapping a Cartesian vector space $\mathbb{E}^2$ (the tangent plane at the point) onto the surface. Locally, this maps a flat image of the locale of the centre onto a geodesic surface (the GeometricReferenceSurface).

NOTE 2 A common example of this usage are the Earth-fixed Engineering Systems (SC_EngineeringCRS) from ISO 19111, most particularly those using polar coordinates, in which all locations are given coordinates based on physical measures (direction and distance) from a fixed point. Once the geographic location of the point is given, and the θ is aligned to a bearing, the exponential map becomes a CRS transform from the Engineering CRS to the coordinate systems using the GeometricReferenceSurface as a datum.

This mapping projects the local tangent space of the GeometricReferenceSurface back to that surface to create local equivalent instances of common Euclidean structures (such as circles) in GeometricCoordinateSystem.

#### 6.2.8.12 Operation: geoLocate

The geoLocate operation maps a position in the geometric coordinate space to the corresponding position in the geocentric coordinates of the embedding space for the geometric reference surface. This function allows algorithms to map complex geographic geometries into classical 3D geometries in a flat 3D Euclidean space using a Cartesian coordinate system.

```
geoLocate(pos:DirectPosition):DirectPosition
```

#### 6.2.8.13 Operation: bearing

This operation "bearing" solves the second geodetic problem that is given two points; find the bearing of the geodesic that joins the first point to the second.

```
bearing(centre:DirectPosition,target:DirectPosition):Bearing
```

NOTE The solution to this problem is only guaranteed to be unique for a small distance. For example, if the two points are antipodal on a sphere, then the bearing between them in indeterminate and any bearing is valid. The only geometric reference surface for which the solution is always unique is a plane; for example, a geometric coordinate system in a Mercator projection.

#### 6.2.8.14 Operation: csInnerProduct

The coordinate system inner product is a function that, for each tangent space defined by the centre variable, defines a "dot product" for the local tangent space. The inner product is useful for doing vector measures such as length, $\left\langle \vec{x},\vec{x} \right\rangle = \left\| \vec{x} \right\|^2$ and angle, $\left\langle \vec{x},\vec{y} \right\rangle = \left\| \vec{x} \right\| \left\| \vec{y} \right\| \cos\theta$ where $\theta$ is the angle in the tangent space between $\vec{x}$ and $\vec{y}$.

```
csInnerProduct(centre:DirectPosition,v1:Vector,v2:Vector):Number
```

NOTE The usual equations for inner products depend on the orthogonality of the axes in the vector space.

### 6.2.9 Datatype DirectPosition

#### 6.2.9.1 Semantics

Instances of the DirectPosition datatype hold the coordinates for a position within some coordinate system (Figure 6), which may be compound and represents an ordered sequence of constituent coordinate systems. Each coordinate system may be associated with at most one spatial coordinate reference system as described in ISO 19111.

#### 6.2.9.2 Attribute rsid

The attribute rsid is a reference by identify to reference system identifiers (R**SID**) of the various components of the coordinate system used for this direct position. Each RSID corresponds to a subset of coordinate array for the axis in the corresponding reference system.

```
DirectPosition::rsid:RSID [1..*] {ordered}
```

#### 6.2.9.3 Attribute dimension

The attribute "dimension" is the dimension of the associated Reference System, and is thus the dimension of the position coordinate.

```
DirectPosition::dimension:Integer
```

NOTE The use of a homogeneous weight (w) would not be counted as a reference system dimension.

Since each component ReferenceSystem occupies its "dimension" offsets, the dimension of the DirectPosition instance is the sum of the dimensions of its ReferenceSystem (corresponding to the list of the RSID in the attribute rsid), giving the following constraint:

```
coordinate.length=SUM(referenceSystem.dimension)
```

#### 6.2.9.4 Attribute coordinate

The attribute "coordinate" is a sequence of Real numbers that hold the coordinates values for this position in the specified reference systems.

```
DirectPosition::coordinate:Real[1..*]
```

#### 6.2.9.5 Derived Role referenceSystem

The association role "referenceSystem" is the sequence of coordinate systems, indexed by the RSID values in the rsid array from 6.2.9.2. Normally, the first in the array will be a realization of the type SC_CRS as described in ISO 19111.

```
DirectPosition.referenceSystem[]:ReferenceSystem
```

The attribute array "rsid" contains the identities of the Reference systems in the derived association role referenceSystem.

```
DirectPosition→referenceSystem[i].rsid = DirectPosition.rsid[i]
```

#### 6.2.9.6 Operation: spatialProjection

**REQ. 32** **The operation "spatialProjection" shall project the DirectPosition to its purely spatial components.**

**Rec. 6** **The resulting DirectPosition should be in an ISO 19111 coordinate reference system.**

```
DirectPosition::spatialProjection():DirectPosition
```

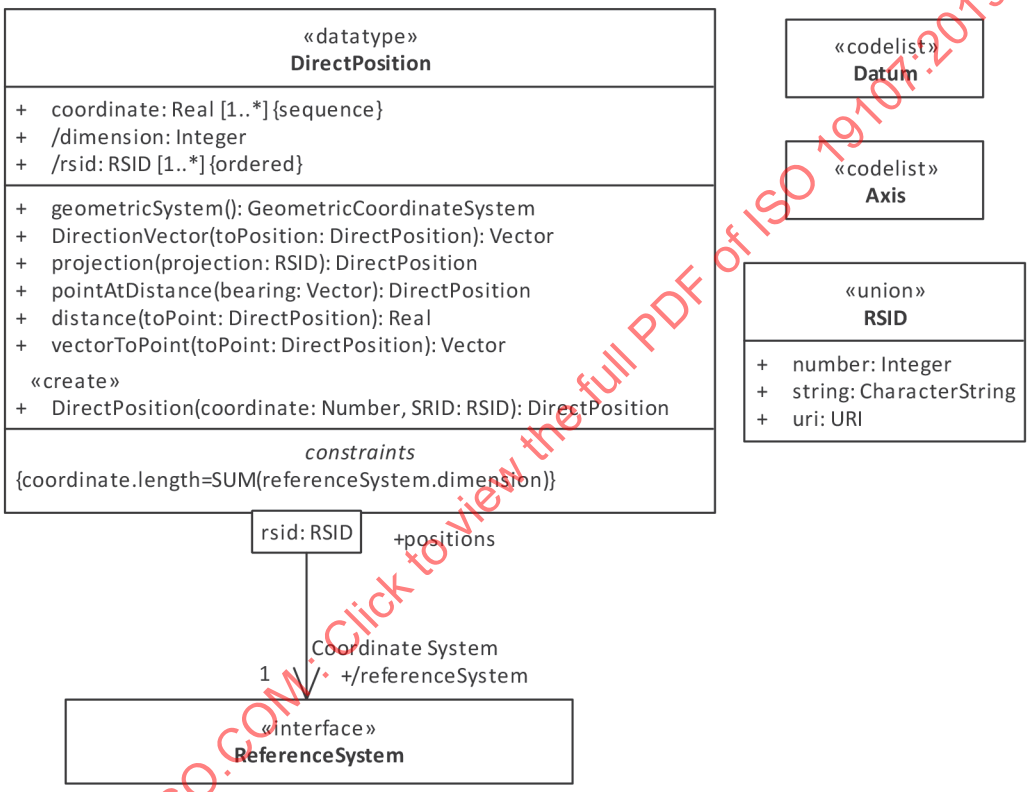



**Figure 6 — DirectPosition**

#### 6.2.9.7 Operation: pointProjection

**REQ. 33** **The operation "pointProjection" shall project the DirectPosition to its purely spatial components, except that it shall still be in homogeneous form, if the original DirectPosition was homogeneous.**

**Rec. 7** **The resulting DirectPosition should be in an ISO 19111 coordinate reference system.**

```
DirectPosition::spatialProjection():DirectPosition
```

NOTE If the original spatial coordinate was in homogeneous form, then if the usually non-homogeneous coordinates, were (x, y, z), the homogeneous form would be *(wx, wy, wz, w)*. The "w" is the weight.

#### 6.2.9.8 Operations derivable from Point by type coercion

##### 6.2.9.8.1 Semantics

The default constructor for Point (see 0) is a complete DirectPosition with the coordinate system information. Therefore, any operation of Point is available for DirectPosition by t**ype coercion.** The other way, the Point attribute called DirectPosition inverts the constructor and returns the "Position" as an instance of DirectPosition.

```
Position.PointMethod(…)=Point(Position).PointMethod(…)

Point.PositionMethod(…)=Point.directPosition.PositionMethod(…)
```

The following two operations are optional in that they are not needed for the principle purpose of DirectPosition, i.e. the indication of location. In a system that implements Point (see 6.4.13), these functions could be replaced by casting the DirectPosition to Point and then executing them from the Point instance. Because these operations are valuable in navigating non Euclidean Figures of the Earth (all but the planes), being able to bypass a Point instance constructor is worthwhile.

**Rec. 8** **Compliant implementations that support the interface Point, or do geometric analysis on a non-planar Reference Surface should support the operations "vectorToPoint" and "pointAtDistance" for DirectPositions.**

Implementation standards dependent on this document may require these operations on direct positions.

##### 6.2.9.8.2 Operation: vectorToPoint (toPoint: DirectPosition): Vector

The operation "vectorToPoint" will return a vector in the tangent space at the point whose direction determines a geodesic curve that intersects "toPoint" DirectPosition at a distance equal to the length of the vector. This operation solves the second geodetic problem.

```
DirectPosition::vectorToPoint
(toPoint:DirectPosition):Vector
```

##### 6.2.9.8.3 Operation: pointAtDistance (bearing: Vector): DirectPosition

The operation "pointAtDistance" will return a DirectPosition given a vector in the tangent space at the point whose direction determines a geodesic curve that intersects that DirectPosition at a distance equal to the length of the vector. This operation solves the first geodetic problem.

```
DirectPosition::pointAtDistance(bearing:Vector):DirectPosition
```

### 6.2.10 Union Datatype RSID

The RSID, or reference system identity, is an identifier for a reference system and a generic ID that can hold a variety of identity types for local use. The RSID content can be a number, a character string or a URI. It is meant to be a local identity, but can be equal to global values from a recognized Authority.

**REQ. 34** **The use of an RSID equal to a global identity linked to a recognized authority shall imply that the local object is consistent with the item defined by that recognized authority.**

### 6.2.11 Codelist Axis

The local instances of the Axis codelist (Figure 7) will contain the names of any axis that can be used in local instances of Reference System. Any name in this codelist will have a class of Axis Description that contains information on the axis' use.

**REQ. 35** **To keep axis types separate, the Axis codelist shall contain sub-list for each general type of axis, including: Spatial, Temporal and Parametric**

### 6.2.12 Role metadata: AxisDescription

Every axis is associated with a metadata description. The exact mechanism for this connection is an implementation option. The simplest mechanism is to use a standard name with a well-known definition (latitude, longitude).

```
Attribute Axis::metadata:AxisDescription
```

### 6.2.13 Datatype Axis Description

Each Axis type will have a metadata description (identified by the name of the Axis).

**REQ. 36** **Each axis in use shall have an Axis Description associated with it by name. The Axis Description instance shall have a URI pointer that identifies all information needed to use the axis in question in a Geometric Coordinate System.**

### 6.2.14 Codelist SpatialAxis

The SpatialAxis codelist will contain all axis names used to represent spatial extent where spatial geometry is described. Each name should be used in at least one of the SC_CRS instance in local use as defined in ISO 19111.

**REQ. 37** **The local instance of the Codelist SpatialAxis shall list all spatial axis types supported by the application.**

### 6.2.15 Codelist SphericalAxis

The SphericalAxis codelist will contain all axis names used to represent spatial extent measured in angles, such as zenithAngle, bearing and radius, used in a polar coordinate system. Each name should be used in at least one of the SC_CRS instance in local use as defined in ISO 19111.

**REQ. 38** **The local instance of the Codelist SphericalAxis shall list all spherical axis types supported by the application.**

### 6.2.16 Codelist TemporalAxis

The TemporalAxis codelist will contain all axis names used to represent temporal extent. Each name will be used in a temporal reference system in a TM_ReferenceSystem value ISO 19108 in local use.

**REQ. 39** **The local instance of the Codelist TemporalAxis shall list all temporal axis types supported by the application.**

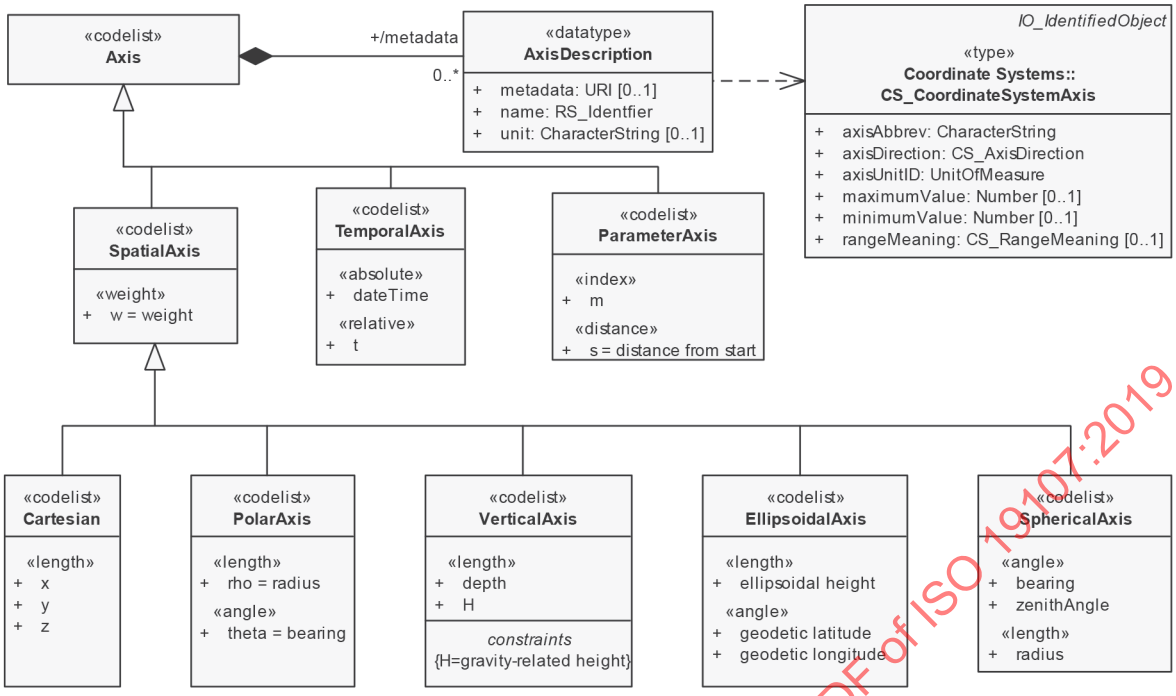



**Figure 7 — Axis**

### 6.2.17 Codelist ParametricAxis

The ParametricAxis codelist contains all axis names used to represent parameters. Each name will be used in one or more parametric reference system in local use as defined in ISO 19111.

**REQ. 40** **The local instance of the Codelist ParametricAxis shall list all parametric axis types supported by the application**

### 6.2.18 Codelist Datum

The local instances of the Datum codelist contains the names or RS_Identifier values from Datums as defined in ISO 19111. Tracing the identifier name back to the data links the local datum to their definitions (Figure 8).

**REQ. 41** **The local instance of the Codelist Datum shall list all Datums or Datum types supported by the application.**

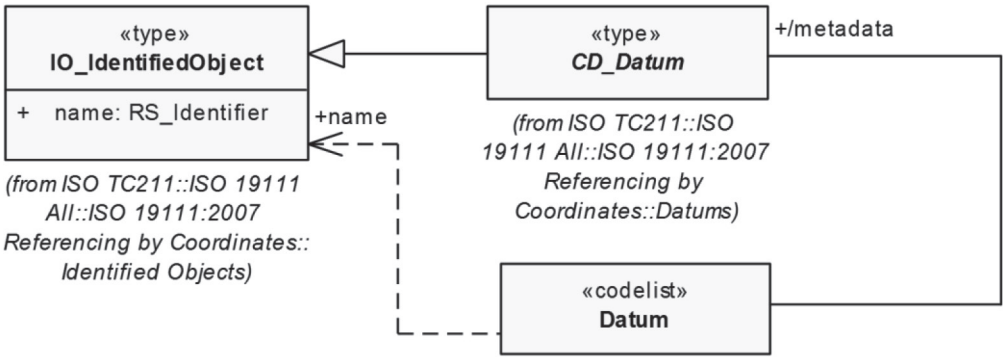



**Figure 8 — Datum**

### 6.2.19 Datatype Parameter

#### 6.2.19.1 Semantics

The datatype Parameter can hold any parameter name and an assigned value for use in any Reference System.

**REQ. 42** **The "m" axis shall be used for "measure parameter" to be consistent with the earlier version of 19107. The "w" axis shall be used if the coordinates are homogeneous.**

**REQ. 43** **In a direct position, all offsets before "w" shall be multiplied by "w" in the transform to homogeneous coordinates.**

Normally, homogeneous coordinates are only used for classical projective geometry purposes, and so the "w" will only affect the spatial projections. Remember that the spatial axes are multiplied by the "w" also.

$$(x,y)\leftrightarrow(x,y,1)\leftrightarrow(wx,wy,w)$$
$$(x,y,z)\leftrightarrow(x,y,z,1)\leftrightarrow(wx,wy,wz,w) \quad (9)$$

#### 6.2.19.2 Attribute name

The attribute "name" carries the name of the parameter.

`Parameter::name:CharacterString`

#### 6.2.19.3 Attribute value

The attribute "value" carries the value of the parameter.

`Parameter::value:Any`

### 6.2.20 Datatype Permutation, Projection

#### 6.2.20.1 Semantics

The datatypes for permutation and projection (Figure 9) are essentially identical except for their usage. A projection selects offsets in the coordinate axis and is then used to create a smaller DirectPosition by limiting the position information to the chosen offsets. A permutation selects offsets in some order and essentially projects to the same space, but with a rearrangement of the offsets. They hold the target positions of the input order. Positions are numbered one up (position as ordinal number, first, second …); Therefore an n-tuple in original order is (1, 2…, n). The projections can select the columns

and reorder what is output. The only difference between the two names is the exhaustive nature. A Permutation has a target coordinate offset for every source coordinate offset (input and output are the same length). A projection can reduce the number of offsets, and is a permutation if its target size is the same as its input size.

#### 6.2.20.2 Attribute outOrder

The outOrder attribute contains the target ordering by listing the output column in order (using ordinate numbering 1=first, 2=second, etc.), by using the offset for pre-image input column.

```
outOrder: Integer [1..*]
```

EXAMPLE If the columns of a 3-tuple are rearranged by 1→3, 2→1 and 3→2 (circular shift to the left); then the representation of the permutation is the sequence (2, 3, 1), i.e. (x, y, z) → (y, z, x). If (lat, long, height) needs to be (long, lat, height) then the permutation is (2, 1, 3).

#### 6.2.20.3 Attribute axis

This is an alternate and parallel of outOrder. The axis list is by name, outOrder is by axis ordinal position. Various encoding can use either one.

```
axis: Axis [1..*]
```

### 6.2.21 Interface ReferenceDirection

The ReferenceDirection interface is empty, but is to be "implemented" by any datatype that can represent a direction (or unit tangent vector) at a point. This lead to a circular, but valid, potentially recursive, definition for the datatype Bearing.

### 6.2.22 Datatype Bearing

#### 6.2.22.1 Semantics

The datatype Bearing denotes a direction. The bearing can take either one of two forms:

— A set of angles, one a planar bearing, and the second a measure of altitude (positive above the horizontal, negative below the horizon), essentially creating a local spherical coordinate system.

— A derived directional vector from the coordinate system at the initial point, a unit vector with the direction associated to the set of angles.

Since the two types of measures are identical, the values of bearing are the same, but the interpretations will depend on the usage (usually a reference direction, the "0" offset and a direction of rotation, clockwise or counter clockwise).

The value of the datatype Bearing may be valid only at the point from which the measure is taken. The common "parallel" transformation of vectors from point to point is only valid if the GeometricReferenceSurface in use is planar (i.e. Cartesian and hence Euclidean). The fundamental problem is that the sphere does not have a 2D universally valid global coordinate system for its tangent spaces. The use of a fixed direction reference such as "true north" does allow for some translation if the reference does exist and is unique at a position. For example, north is non-existent at the North Pole, and not unique at the South Pole.

Bearings may be measured two ways, absolute (such as true north, see bearing) or relative (such as "fore" or "aft").

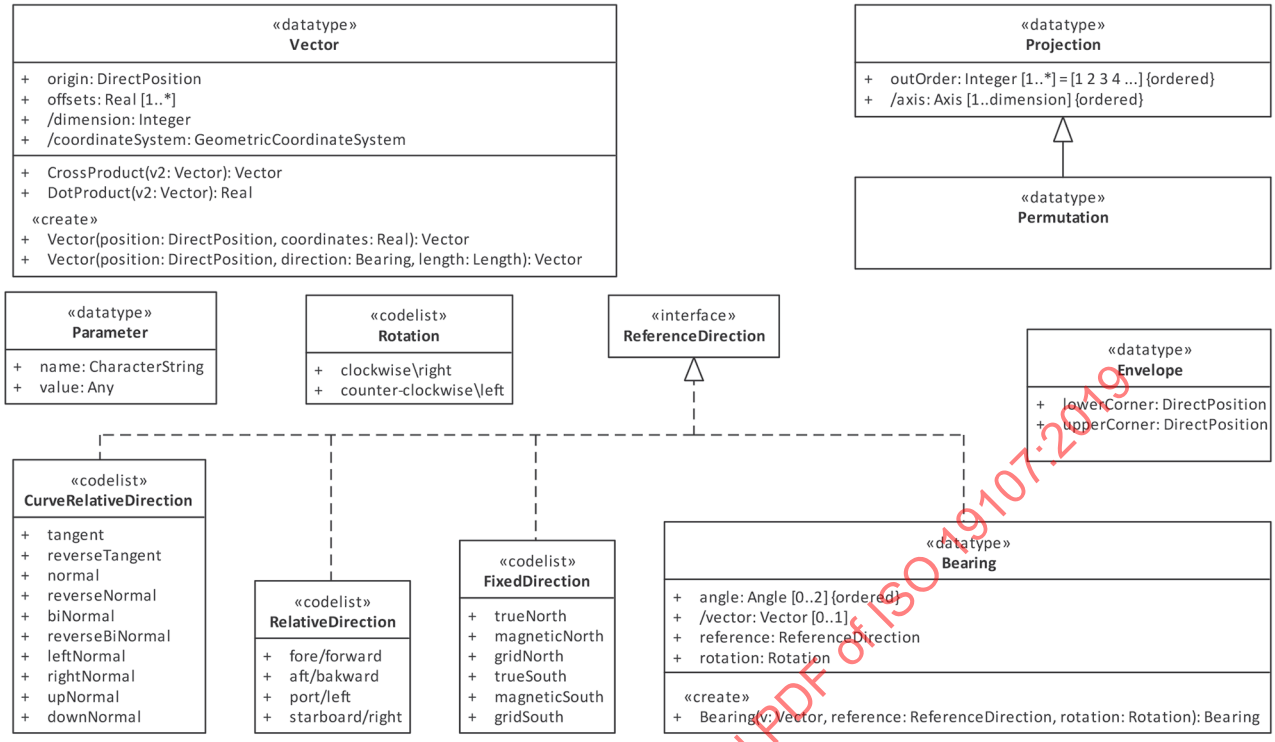



**Figure 9 — Vector, Bearing and Permutation**

**REQ. 44 The reference direction for an instance of Bearing shall not transitively reference itself.**

Bearing is used to represent direction in the coordinate system. In a 2D coordinate reference system, this can be accomplished using an "angle measured from true north" or a 2D vector point in that direction. In a 3D coordinate reference system, two angles or any 3D vector is possible. If both a set of angles and a vector are given, then they shall be consistent with one another. In coordinate systems with higher dimensions, similar representations are possible.

#### 6.2.22.2 Attribute angle

In this variant of Bearing usually used for 2D coordinate systems, the first angle (azimuth) is measured from a coordinate axis (usually north) in a clockwise fashion parallel to the Reference Surface tangent plane. If two angles are given, the second angle (altitude) usually represents the angle above (for positive angles) or below (for negative angles) a local plane parallel to the tangent plane of the Reference Surface.

```
Bearing::angle:Angle[1..*]
```

#### 6.2.22.3 Attribute direction

In this variant of Bearing usually used for 3D coordinate systems, the direction is expressed as an arbitrary vector, in the coordinate system.

```
Bearing::direction:Vector[0..1]
```

#### 6.2.22.4 Attribute reference: ReferenceDirection

The attribute reference is the direction from which the bearing is measured, i.e. the direction of a positive measure. Most systems can use a negative number to express a measure opposite of the default rotation.

```
reference: ReferenceDirection
```

#### 6.2.22.5 Attribute rotation: Rotation

The attribute rotation specifies the direction from which the bearing is measured.

#### 6.2.22.6 Constructor: bearing

The default constructor for bearing takes any non-zero vector $\left(\left\|\vec{v}\right\|>0\right)$ at a point and creates a bearing at that point, with the standard defaults for the most common fixed bearing.

```
Bearing (v: Vector, reference: ReferenceDirection="north",
rotation: Rotation="clockwise"): Bearing
```

If the bearing of a curve $c(t)$ is required, then the vector would be $\dot{c}(t)$ for the position defined by the parameter "$t$". Since speed is not pertinent for direction, the magnitude of the vector has no effect in bearing. This means that if the vector $\vec{v}$ is used in the constructor, the derived value of the vector in the Bearing constructed will be $\vec{v}/\left\|\vec{v}\right\|$, a unit vector in the same direction as the tangent.

### 6.2.23 Codelist Rotation

The codelist Rotation lists the two potential rotational direction for an angular measure, clockwise and counterclockwise. These are as viewed from above the Reference Surface.

### 6.2.24 Codelist RelativeDirection

The codelist RelativeDirection lists the common potential relative reference directions for bearing, usually used in reference to a moving vehicle. The most common values include fore (forward), aft (backwards), port (90° left) or starboard (90° right).

### 6.2.25 Codelist FixedDirection

The codelist FixedDirection lists the common potential fixed reference directions for bearing, usually used in reference to the globe, a map, a coordinate system or a grid. The most common values include true north or south, magnetic north or south, grid north or south (reference to a grid or projection).

### 6.2.26 Codelist CurveRelativeDirection

The codelist CurveRelativeDirection refers to vectors associated to a curve. Some common vector directions are:

— tangent, the direction the curve is pointed $\left(\vec{\tau}=\dot{c}(t)/\left\|\dot{c}(t)\right\|\right)$;

— reverse tangent, opposite of the tangent, $\left(-\vec{\tau}\right)$;

— biNormal, the direction towards the centre of curvature, the inside of the curve;

— reverseBiNormal, opposite of the biNormal, the outside of the curve;

— leftNormal, leftward from the tangent;

— rightNormal, rightward from the tangent;

— upNormal relative the Reference Surface;

— downNormal, opposite of the upNormal.

These valued may be used to either extend or replace the 2D directions from RelativeDirection.

### 6.2.27 Datatype Vector

#### 6.2.27.1 Semantics

The common "vector" in Euclidean spaces may be translated to any point in the space because the flat nature of the Euclidean space has a universal coordinate system that works at all "start" point of a vector.

The datatype Vector in this document must be associated with a point on the GeometricReferenceSurface to be well defined. The directional parts of the vector must also specify the "start position" of the vector. Where the coordinate system is well behaved, the spanning vectors of the tangent space are represented in the 3D geocentric space by the tangents to the coordinate functions. For example, in a latitude, longitude system, a local tangent space basis are the unit tangent vectors to the curves of constant longitude, and the curves of latitude, except for the poles where the longitude function has a zero tangent at the pole. If (*lat, long* = (*dφ, dλ*) then the symbols (*dφ, dλ*) are the differentials of the two coordinate functions represented by tangents in the direction of increasing φ and λ respectively, and represent a basis for the local tangent spaces.

#### 6.2.27.2 Attribute origin

The attribute origin is the location of the point on the GeometricReferenceSurface for which the vector is a tangent. The direct position is associated with a coordinate system; this determines the coordinate system for the vector. The direct position's spatial dimension determines the dimension of the vector.

```
Vector::origin:DirectPosition
```

#### 6.2.27.3 Attribute offset

The attribute offset uses the coordinate system of the direct position and represents the local tangent vector in the differentials of the local coordinates.

```
Vector::offset:Real[1..*]
```

EXAMPLE If the Euclidean plane ( $\mathbb{E}^2$ ) is in use, then the DirectPosition is a coordinate pair (x, y) and the vector is the differentials in both directions, hence offset is two long and has a coordinate base of (dx, dy).

#### 6.2.27.4 Attribute dimension

The attribute dimension is the dimension of the origin and therefore the dimension of the local tangent space of the vector.

```
Vector::dimension:Integer
```

#### 6.2.27.5 Attribute coordinateSystem

The attribute coordinateSystem is the system of the origin and therefore determines the coordinates of the local tangent space in which the vector exists.

```
Vector::coordinateSystem:GeometricCoordinateSystem
```

#### 6.2.27.6 Operations: CrossProduct, DotProduct

The dot product returns a real value that is the sum of the products of the corresponding coefficients of the two vectors. The dot product is a third vector perpendicular to the other two. If the vector space is only two dimensional, the cross product returns a directed magnitude and not a vector.

```
v1.DotProduct(v2:Vector):Number
v1.CrossProduct(v2:Vector):Vector

«create»
Vector (position:DirectPosition, coordinates:Real[]):Vector
Vector (position:DirectPosition, direction:Bearing,length: Length):Vector.
```

The constructor vector creates a vector with the given offsets or the given direction and length, at the specified direct position, in the coordinate space of the direct position.

### 6.2.28 Interface Envelope

#### 6.2.28.1 Semantics

Envelope is often referred to as a minimum bounding box or rectangle. Regardless of dimension, an Envelope can be represented without ambiguity as two DirectPositions (coordinate points). To encode an Envelope, it is sufficient to encode these two points. This is consistent with all of the coordinate systems in this document. Recall, even if the CoordinateSystem is purely spatial but not globally one-to-one, the coordinate may not be valid in the associated coordinate system.

NOTE Envelopes are not always equivalent to "rectangles" since this polygon may not exist as a geometry object if either or both of the two corners is not in the coordinate system or if the 4-point polygon in 2D or 8-point polyhedral in 3D equivalent is not a valid geometry because of coordinate validity issues. Essentially, envelopes are rectangles in $\mathbb{R}^n$.

Mathematically, the definition of an Envelope is:

$$\left[\forall A \in Geometry, \forall p \in DirectPosition\right] \Rightarrow$$
$$\left[\forall i \in \mathbb{Z}, A.Envelop.lowerCorner[i] = \min\{p(i) \mid p \in A\}\right] \wedge \quad (10)$$
$$\left[\forall i \in \mathbb{Z}, A.Envelop.upperCorner[i] = \max\{p(i) \mid p \in A\}\right]$$

Since an Envelope is a set of direct positions, it is also a geometry. Containment is determined by coordinate ranges. Notice that Req10 on periodic coordinate offsets will affect the way some spatial indexes (r-trees in particular, see [36]) will work. For example, a spatial r tree index based on Envelope in latitude, longitude in the ranges (−90, +90) and (−180, 180) may require an Envelope be divided in parts to allow it to be represented in the specified ranges. This would allow an index based on the envelops to be able to use the standard r tree assumption that:

$$\left[A \cap B\right] \Leftrightarrow \left[A.envelope \cap B.envelope\right] \quad (11)$$

**REQ. 45 The envelope of the empty geometry shall be the empty geometry.**

This requirement is forced by the mathematical definitions. The min(∅) = +∞ and max(∅ = −∞. A real number cannot be greater than "+∞" nor smaller than "−∞". The definitions of **max** and **min** are in Clause 3.

#### 6.2.28.2 Attribute upperCorner

The "upperCorner" of an Envelope is a coordinate position consisting of all the maximal coordinates for each dimension for all points within the Envelope.

```
Envelope::upperCorner:DirectPosition
```

of that offset in all direct position in the original geometry.

#### 6.2.28.3 Attribute lowerCorner

The "lowerCorner" of an Envelope is a coordinate position consisting of all the minimal coordinates for each dimension for all points within the Envelope.

```
Envelope::lowerCorner:DirectPosition
```

Math For each coordinate offset, the corresponding offset of the lower corner is the minimum of that offset in all direct positions in the original geometry and. the corresponding offset of the upper corner is the maximum of those same direct positions.

### 6.2.29 Engineering coordinate systems, Tangent spaces and local interpolations

In ISO 19111, the class CD_EngineeringDatum defines "the origins of an engineering coordinate reference system, and is used in a region around that origin". The class SC_EngineeringCRS is a "contextually local coordinate reference system associated with an engineering datum".

The tangent space at a point in a geometric coordinate system is the collection of all tangent vectors (of all lengths) at that point. The coordinate system used for vectors is a $\mathbb{E}^2$ or $\mathbb{E}^3$ (depending on the dimension of the underlying spatial CRS. The expression of bearings can be either polar $(r,\theta)$ or spherical $(r,\Omega,\theta)$ (see ISO 19111). The operation pointAtDistance in the interface GeometricCoordinateSystem, maps tangent vectors at a point to other points (given by a DirectPosition) using geodesic curves computation. This operation (formally called the exponential map in differential geometry) makes the tangent space (a copy of $\mathbb{E}^n$ a Cartesian coordinate space) equivalent to a SC_EngineeringCRS. Standard Euclidean constructions (lines, circles, ellipses, general conics and spirals) can work directly in this local engineering CRS and then projected onto the GeometricReferenceSurface using the exponential map.

This document recognizes two general types of interpolations for geometry: algebraic and geodetic.

Algebraic interpolations will treat the coordinates independent of geometric interpretation, strictly as variables in algebraic equations. Lines, polynomial splines, b splines and solid boundary representations are examples of algebraic interpolations.

Geodetic interpolations will use geometric surface considerations to interpolate direct position values, most commonly using the local tangent space at one of the control points to use Euclidean constructions and then using the exponential map to transfer these constructions to the Reference Surface. Circles, spirals and other geometry types dependent on distance, direction or on Newtonian physics will fall into this second category.

Some design procedures may jump this divide by using parts of both techniques. Linear interpolation in the tangent space starting at the "tangent point" produces geodesics originating from that tangent point. Spirals work in local Engineering spaces, and should be done using the exponential map centred at some critical design point.

NOTE The word "line" was originally used for any rope, thread or string and has the same base as "linen". In mathematics, the word "line" means the use in Euclid's elements, which is normally linear interpolation in a Euclidean space ( $\mathbb{E}^n$ ). In this document, line will be linear interpolation in the current coordinate system.

## 6.3 Requirements Class Coordinate Data

**REQ. 46** **An implementation of the requirements class Coordinate Data shall support all datatypes in Requirements Class Coordinate.**

## 6.4 Requirements Class Geometry

### 6.4.1 Semantics

The geometry packages contain the various classes for geometry. All geometry classes inherit, from the root interface Geometry (Figure 10), an implicit link to a reference system (normally a geometric coordinate system) which defines the meaning of the coordinates in its direct positions. Each primitive geometric object may contain other coordinate offsets, but will always have a constant coordinate system and Reference Surface.

**REQ. 47** **An implementation of the Requirements Class Geometry shall implement Requirements Class Coordinate.**

**REQ. 48** **An implementation of the Package Geometry shall have all instances and properties specified for this package, its contents and its dependencies, contained in the UML model for this package in this document.**

**REQ. 49** **All direct positions exposed through the geometry object interfaces shall be in the coordinate reference system of the geometric object accessed unless that operation protocol specifically specifies a different return reference system.**

**REQ. 50** **All elements of a geometric collection shall be in the same coordinate reference system as specified by that collection.**

Figures 10, 11, 12 and 13 show the basic behavioural classes and datatypes defined in the geometry packages. Any object that inherits the semantics of Geometry acts as a set of Direct Positions. Its behaviour depends solely on which Direct Positions it contains.

Objects under geometry will be interpreted as metrically closed; any point whose distance to the geometry object is zero is on the object ("in" when the object is viewed as a set of direct positions). Curves will contain their endPoints; surfaces will contain their boundary curves, and solids will contain their bounding surfaces.

NOTE The use of containment prepositions "in" or "on" for geometry is equivalent and can be used interchangeably. "On" is probably more appropriate when speaking of geometry as a location reference, but "in" sounds better when speaking of geometry objects as sets of positions.

When a primitive closes back on itself, a curve starting and ending at the same point, or a surface which folds back onto itself and is thus on "both sides" of all of its boundary curves, it is called a cycle, and is normally the boundary of a geometry of one higher dimension.

**REQ. 51** **In all of the operations, all geometric calculations shall be done in the coordinate system of the first geometric object accessed, which is normally the object whose operation is being invoked.**

**REQ. 52** **Returned objects shall be in the coordinate system in which the calculations are done unless explicitly stated otherwise by the semantics of the operation.**

Many of the protocols defined in this section are those of set theory. In general, a geometric object can be viewed as a set (usually infinite) of geometric points, each representable by a DirectPosition.



**REQ. 53** **Geometry instantiations of geometric objects shall realize the Geometry interface. Geometry instantiations of geometric points, when used as values, shall realize DirectPosition.**

**REQ. 54** **Geometry and Geometry Primitive are abstract in the sense that no object or data structure from an application schema can instantiate them directly.**

**REQ. 55** **Instances of Geometry and Geometry Primitive shall also be instances of one of one of their dimension specific subtypes, such as Point, Curve, Surface or Solid.**

This is not the case for Geometry Complex, which can be directly realized by an application schema class. Although Geometry Complex is not explicitly implemented by this document, it would be valid for an application schema to include a concrete class called "Geometry Complex" in a class library conformant to this document. Recall that the name space of the application schema is different from that of this document and such seemingly logical abuses of name are nonetheless valid. This is not the case for the purely abstract interfaces within this document. These interfaces are logically incapable of supporting an implementation directly. Constructors on these interfaces result in instances of more concrete subtypes, not in direct logical instances of the root type.

### 6.4.2 Interface TransfiniteSetOfDirectPositions

Many geometry operations are simple set theory, which does not vary based on the cardinality of the sets involved. Unfortunately, the term "set" in most programming languages refer to a finite collect of objects or object identities. The abstract interface "TransfiniteSetOfDirectPositions" represents the set theory operations that cannot be always be tested by simple enumeration techniques for sets with a small number of members.

A "TransfiniteSetOfDirectPositions" can be very large in the number of elements (limited only to finite by the use of finite precision digital computing). The collection

$$\{x \in \mathbb{R} \mid 0 \le x \le 1\}$$

is logically infinite. On a digital computer it may only have $2^{64}$ things in it, big but not infinite. "Transfinite" represents that nothing on a digital computer is truly infinite, but that does not keep it from being too large to deal with in reasonable time.

The sole operation is a testing operation "contains", which applies whatever hidden function is required to determine whether a particular "direct position" is in the set. Each Geometry Interface type will have to implement this with the limitations of a limited accuracy computing platform.

If A is a TransfiniteSetOfDirectPositions, then:

$$\left[\boldsymbol{x} \in \boldsymbol{A}\right] \Rightarrow \left[A.contains\left(\boldsymbol{x}\right) = \boldsymbol{TRUE}\right] \tag{12}$$

### 6.4.3 CodeList: BoundaryType

The codelist BoundaryType (Figure 2) describes how boundary of a set is calculated in some "ambiguous" situations. Common values are as follows:

— metric (from metric spaces, based on distance and topology);

— mod 2;

— at least 2.

The metric boundary comes from metric spaces, which generalize $\mathbb{R}^n$ spaces. A metric space A is any set that has a function $m: A \times A \to \mathbb{R}$ that acts like a distance. Formally, this means "m" satisfies three axioms:

$$\begin{aligned} & m: A \otimes A \to \mathbb{R} \Leftrightarrow \\ & \left[\forall x, y \in A, m(x,y) = m(y,x)\right] \wedge \\ & \left[\left(\forall x, y \in A, m(x,y) = 0\right) \Leftrightarrow x = y\right] \wedge \\ & \left[\forall x, y, z \in A, m(x,z) \leq m(x,y) + m(y,z)\right] \end{aligned} \tag{13}$$

Given any metric, we can define a distance between a point $x \in$ M and a subset A of M.

$$\boldsymbol{dist}(\boldsymbol{x}, \boldsymbol{A}) = \boldsymbol{min}\left\{\boldsymbol{m}(\boldsymbol{x}, \boldsymbol{a}) \,|\, \boldsymbol{a} \in \boldsymbol{A}\right\} \tag{14}$$

The definition of boundary is given by:

$$\left[\forall \boldsymbol{x} \in \boldsymbol{M}\right] \Leftrightarrow \left[\left[\boldsymbol{x} \in \partial \boldsymbol{A}\right] \Leftrightarrow \left[\boldsymbol{dist}(\boldsymbol{x}, \boldsymbol{A}) = \boldsymbol{dist}(\boldsymbol{x}, \boldsymbol{M} - \boldsymbol{A}) = 0\right]\right] \tag{15}$$

The "mod 2" and at "least 2" use the Primitives definition of their boundary, but disagree on how Collections work. This is described in detail in 10.8.2.

### 6.4.4 Interface Geometry

#### 6.4.4.1 Semantics

Geometry (Figure 10) is the root class of the geometric object taxonomy and supports interfaces common to all geographically referenced geometric objects. Geometry instances are sets of DirectPositions in a particular coordinate reference system. Geometry can be regarded as a potentially infinite set of points that satisfies the set operation interfaces for a set of DirectPositions that is a TransfiniteSetOfDirectPositions (essentially a set defined by a Boolean "isIn" operator). Since an infinite collection class cannot be implemented directly, a Boolean test for inclusion shall be provided by the Geometry interface. This document concentrates on vector geometry classes, but future work may use Geometry as a root class without modification.

NOTE As a type, Geometry does not have a well-defined default state or value representation as a data type. Instantiated subclasses of Geometry will have.

Attributes are values assigned to represent properties of an object. If the general content of the object is such that the value might be calculable from a reasonable representation of the object, this document has marked it as "derived". There is no functional difference between an attribute and an operation without parameters. In «interface» classifiers in UML, the mechanism of implementation is undefined.

**Per. 1** **Implementations may support any attribute or operation from this document in any manner, as derived or stored attributes or as functions with or without parameters.**

**REQ. 56** **The use of geometry instance for the description of features shall be consistent with ISO 19109.**

**REQ. 57** **All attributes in the root class Geometry, except isEmpty, shall be consistent with the basic attribute defined in the default constructor's type, all of which are subtype of the datatype GeometryData.**



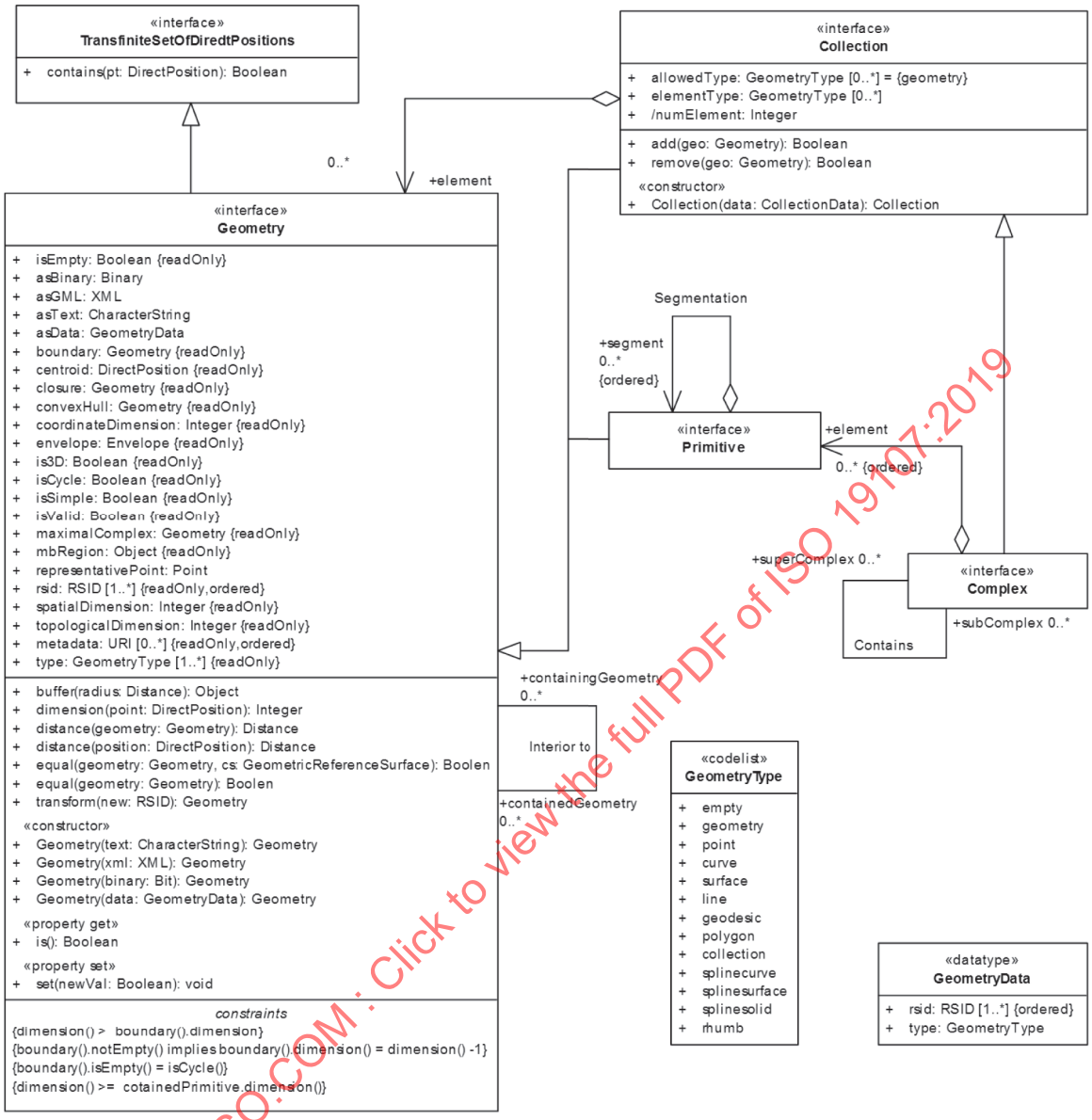



**Figure 10 — Geometry root object**



### 6.4.4.2 Attribute isEmpty

The Boolean valued attribute isEmpty indicates that the instance of geometric object is the empty set. Since the empty set is unique, all empty objects are "spatially equal" to any other empty object. Once an instance is known to be empty *(this.IsEmpty = TRUE)*, the rest of the information in the object becomes redundant until the *this.isEmpty* Boolean is reset to FALSE.

```
isEmpty: Boolean
```

If the isEmpty is set to TRUE, the object is locked until the isEmpty is set to FALSE. Essentially setting the isEmpty to TRUE changes the behaviour (as defined by its class) of the object to match the class Empty, defined in 6.4.10. The implementation of this “class” or “behaviour” change may vary depending programming language or other aspects of the object orientation mechanism.

**Per. 2** **Implementation may use the isEmpty Boolean to set objects to Empty temporarily until a "transaction" or similar action is either completed or undone.**

**Rec. 9** **Long-term existence (persistence) of an "empty" geometry is not recommended but is not prohibited.**

NOTE In this manner, "isEmpty" can be used much as "marked for deletion" is in some drafting systems, allowing a geometry to be kept for "undo" commands without its appearance in the display.

**REQ. 58** **Any geometric object for which isEmpty=TRUE, shall behave appropriately in its other operations and attributes; e.g., this.boundary, this.centroid and other derived Geometry are all empty, and any derived DirectPosition shall return a NULL value or flag.**

**REQ. 59** **The topological dimension of an empty geometry shall be –1.**

There is only one empty geometry. Multiple instances may represent the empty geometry, but all are spatially equal.

**REQ. 60** **All empty geometry objects shall be spatially equal.**

**REQ. 61** **If isEmpty=TRUE, the GeometryType="empty".**

#### 6.4.4.3 Attribute asBinary

The attribute asBinary will be a well-known binary (WKB) representation for the Geometry.

`asBinary:Binary`

#### 6.4.4.4 Attribute asGML: XML

The attribute asGML will be a GML representation for the Geometry.

`isGML:XML`

#### 6.4.4.5 Attribute asText: CharacterString

The attribute asText will be a well-known text (WKT) for the Geometry.

`asText:CharacterString`

#### 6.4.4.6 Attribute asData: GeometryData

The attribute asData will be the default datatype for the constructor for this element

`asData:GeometryData`

#### 6.4.4.7 Attribute boundary: Geometry

The “boundary” attribute is the topological boundary of this Geometry. The boundary will always have a topological dimension one smaller than the current object, unless the object is empty or is a cycle ("isCycle=TRUE").

`boundary:Geometry`

**REQ. 62** **The finite set of Geometries returned by boundary shall be in the same coordinate system as this Geometry.**

The boundary Geometry Geometries returned may have been constructed in response to the operation.

**REQ. 63** **The return value of the boundary of a cycle ("isCycle=TRUE") shall always be an instance of Empty.**

**REQ. 64** **The return value of the boundary of a point shall always be an instance of the empty set as a geometry object. For every instance of the point interface or of any representation of the empty geometry shall also have "isCycle=True".**

**REQ. 65** **The boundary of a curve if not empty or not a cycle shall be 2 points: the first (startPoint) and last (endPoint) point of the curve.**

**REQ. 66** **The boundary of a surface, if not empty or a cycle, shall be a set of curves, each of which is a cycle and is simple and each of which has the surface on its left, but not its right.**

**REQ. 67** **The boundary of a solid, if not empty, shall be a set of surfaces, each of which is a cycle.**

**REQ. 68** **In a 2D spatial system, the only surface which is a cycle (having an empty boundary) shall be the universal surface containing all possible positions or the empty set.**

**REQ. 69** **In a 3D spatial system, the only solid which is a cycle (having a empty boundary) shall be the universal solid covering all possible positions or the empty set.**

The organization of the set returned is dependent on the type of Geometry. Each instantiation of the subclasses of Geometry described below may specify the organization of its boundary set more completely dependent on the level to which the application has conformant implementations of Geometry Complex or Topology Complex.

**REQ. 70** **All objects in the boundary of a geometry shall be of at least 1 dimension smaller than the dimension of the original object.**

**REQ. 71** **If a non-empty, non-cyclic geometry object shall be a primitive other than a point, then the boundary geometry object is 1 dimension smaller than the dimension of the object. The boundary of a finite set of points shall be empty.**

NOTE Some topology text use "–1" as the dimension of the empty set to make points just another example of the above.

#### 6.4.4.8 Attribute centroid

The attribute centroid is a coordinate location that is the "centre of gravity" of the geometry.

```
centroid:DirectPosition
```

The centroid is a calculated value and may not be actually be on the geometric object.

**REQ. 72** **The attribute centroid shall return the position of the geometric centroid of the geometric object.**

For heterogeneous collections of primitives, the centroid only takes into account those of the largest dimension. For example, when calculating the centroid of surfaces, an average is taken weighted by area. Since curves have no area, they do not contribute to this average.

EXAMPLE A "perfect" donut (torus) has its centroid in its "hole in the centre".

There can be cases for which this position would be outside the domain of validity of the object's coordinate reference system, but this is unlikely, since the domain of validity of most coordinate reference systems is convex. If this unlikely case should arise, the implementation shall decide on appropriate action. Modifications of the centroid algorithm are possible to interpret curves as centrelines associated with some width, or points as centres with radius. In such cases, the dimensional restriction for mixed aggregates may not hold. The implementation specification shall decide on appropriate interpretations base on their requirements.

#### 6.4.4.9 Attribute closure

**REQ. 73** **The attribute "closure" shall return a finite set of Geometries containing all of the points on the boundary of this Geometry and this object (the union of the object and its boundary).**

These object collections shall have further internal structure where appropriate. The finite set of Geometries returned shall be in the same coordinate reference system as this Geometry. If the Geometry is in a Complex, then the boundary Geometries returned shall be in the same Complex. If the Geometry is not in any Complex, then the Geometries returned may have been constructed in response to the operation.

```
closure:Geometry
```

#### 6.4.4.10 Attribute convexHull

```
convexHull:Geometry
```

**REQ. 74** **The attribute convexHull shall return the smallest convex set containing the geometric object.**

There can be cases for which this calculated convex Geometry would be partially outside the domain of validity of the object's coordinate reference system, but this is unlikely, since the domain of validity of most coordinate reference systems is convex. If this unlikely case should arise the implementation shall decide on appropriate action. Convexity requires the use of "lines" or "curves of shortest length" and the use of different coordinate systems may result in different versions of the convex hull of an object. Each implementation shall decide on an appropriate solution to this ambiguity. For two reasonable coordinate systems, a convex hull of an object in one will be very closely approximated by the transformed image of the convex hull of the same object in the other.

#### 6.4.4.11 Attribute coordinateDimension

The attribute coordinateDimension is the number of axes in the reference system.

```
coordinateDimension:Integer
```

#### 6.4.4.12 Attribute envelope

The envelope of a geometric object is the smallest coordinate rectangle containing the geometric object.

**REQ. 75** **The attribute "envelope" shall return the smallest coordinate rectangle in the corresponding reference system that contains all the direct position in the geometry object.**

**REQ. 76** **Any coordinate tuple in the data points shall be interior to the "envelope", with respect to inequalities defining the envelope, the coordinate form it is presented in. (see REQ.10).**

The "envelope" will be the minimum bounding box for this Geometry. This will be the coordinate region spanning the minimum and maximum value for each coordinate offset taken on by DirectPositions in this Geometry. The simplest representation for an envelope consists of two DirectPositions, the first one containing all the minimums for each coordinate offset, and second one containing all the maximums. However, there are cases for which these two positions would be outside the domain of validity of the object's coordinate system.

```
envelope:Envelope
```

NOTE If a coordinate system "wraps" around singularities, there can be several alternate representations for the envelope. Applications that allow this will have to be careful in choosing one that makes sense in their operational concept. This document does not specify how this is done.

#### 6.4.4.13 Attribute is3D: Boolean

```
Geometry::is3D:Boolean
```

The Boolean attribute "is3D" indicates where the geometric object is using three spatial dimensions.

**REQ. 77** **If "is3D" is TRUE for an instance of Geometry, then the spatialDimension of that instance is 3.**

#### 6.4.4.14 Attribute isCycle

```
isCycle:Boolean
```

The Boolean attribute "isCycle" indicates whether the boundary is empty. The common term often used is "closed" but that has a specific topological meaning, and this document uses "isCycle" to prevent confusion.

**REQ. 78** **The attribute "isCycle" shall be true if and only if the boundary of the geometric object has an empty boundary.**

The attribute "isCycle" will be TRUE if this Geometry has an empty boundary after topological simplification (removal of overlaps between components in non-structured aggregates, such as some subclasses of Collection). This condition is alternatively referred to as being "closed" as in a "closed curve". This creates some confusion since there are two distinct and incompatible definitions for the word "closed". The use of the word cycle is rarer (generally restricted to the field of algebraic topology), but leads to less confusion. Essentially, an object is a cycle if it is isomorphic to a geometric object that is the boundary of a region in some Euclidean space. Thus, a point is a cycle as its boundary is empty. A curve is a cycle if it is isomorphic to a circle (has the same start and end). A surface is a cycle if it is isomorphic to the surface of a sphere, or some torus (possibly with some number of holes – i.e. of a genus not 1). A solid with finite size in a coordinate space of dimension 3 is never a cycle. For a finite solid to be a cycle, it has to exist in four dimensions.

EXAMPLE The following OCL uses the boundary to view Geometry and then tests for an empty set using the isEmpty.

```
isCycle ⇔ boundary.isEmpty
```

#### 6.4.4.15 Attribute isSimple

```
isSimple:Boolean
```

The Boolean attribute "isSimple" indicates whether the subject self-intersects (either by self-crossing or by self-tangency).

**REQ. 79** **The attribute isSimple shall returns FALSE if and only if the geometric object has a point of self-intersections or has a point of self-tangency.**

The attribute "isSimple" will be TRUE if this Geometry has no interior (non boundary) points of self intersection or self tangency. In mathematical formalisms, this means that every point in the interior of the object must have a metric neighbourhood whose intersection with the object is topologically isomorphic to an n sphere, where n is the topological dimension of this Geometry.

Since most coordinate geometries are represented, either directly or indirectly by functions from regions in Euclidean space of their topological dimension, the easiest test for simplicity to require that a function exists that is one-to-one and bicontinuous (continuous in both directions). Such a function is a topological isomorphism. This test does not work for "closed" objects (that is, objects for which the isCycle operation returns TRUE) which will not be one to one on the boundaries of the parameterization domain.

While Geometry Complexes contain only simple Geometry Geometries, non simple Geometries are often used in "spaghetti" data sets.

"Spaghetti" is a pejorative (uncomplimentary) term, usually indicative of an unacceptable level of geometric anomalies and inconsistencies in the data that must be "cleaned" before use is made of it. Such inconsistencies can include (but are not limited to) any or all of the following anomaly types: An undershot line is a line that should intersect another, but falls short leaving a small gap between it and the point of intersection. This is often hard to distinguish from real "near misses" between features (such as where a road is separated from another by a wall only one brick thick). This problem is especially difficult to handle when the undershoot fails to close a surface or polygon boundary. This is often indicative of the digitizer working at too small a scale and failing to "snap" to the end of lines. An overshot line is a line that should intersect and terminate at another, but goes too far, leaving a small excess line on the far side of the point of intersection. This is often indicative of the digitizer working at too small a scale and trying to "snap" the end of lines. End loop (a line that should intersect and terminate at another, but goes too far and then returns, leaving a small excess loop on the far side of the point of intersection. This is often indicative of the digitizer working at too small a scale and "snapping" a line after he has already overshot it. Slivers and gaps are multiple lines that should represent the same geometry, but do not coincide, leaving areas of overlap between two surface boundaries (slivers) and gaps between them. This problem is particularly difficult to deal with in areas of braided streams where the real geometry of the natural feature resembles the sliver and gaps of simple bad digitization practice. This is often indicative of multiple sources for the same data, which have been merged (but not properly conflated), into the same database. The real problem with "spaghetti" comes in that the heuristics (either manual or automated) used to correct the problems often result in additional but different factual errors. This can be a severe quality issue for geometry.

#### 6.4.4.16 Attribute isValid

The Boolean attribute "isValid" tests to assure that the local representation of geometry is valid.

```
isValid:Boolean
```

**REQ. 80** **The attribute "isValid" shall be TRUE if and only if the structure of the geometric object is valid as geometry.**

#### 6.4.4.17 Attribute maximalComplex

As a set of primitives, a Geometry may be contained as a set in another larger Geometry, referred to as a "super set" of the original. A Geometry is maximal if there is no such larger super set in the same dataset.

**REQ. 81** **The attribute "maximalComplex" shall return the maximal set of Geometry within which this Geometry is contained.**

```
maximalComplex:Geometry
```

If the application schema used does not include Complex, then this operation may return a NULL value.

NOTE The usual semantics of maximal complexes does not allow any Primitive to be in more than one maximal complex, making it a strong aggregation. This is not an absolute, and depending on the semantics of the implementation, the association between Primitives and maximal Complexes could be many to many. From a programming point of view, this would be a difficult (but not impossible) dynamic structure to maintain, but as a static query only structure, it could be quite useful in minimizing redundant data inherent in two representations of the same primitive geometric object.

#### 6.4.4.18 Attribute metadata: URI[0..*]

The optional attribute metadata is a list of URIs (Uniform Resource Identifiers) of references to documentation that gives information on the geometry type (class or interface) for the implemented geometry. If the value of metadata is NULL, or if the metadata attribute is not supported, then the necessary information is in this document.

```
metadata: URI[0..*]
```

The attribute metadata may be a static attribute of the implementation class of this object.

**REQ. 82** **Each value of the attribute Geometry:metadata shall be URI references to information on the implementation of this geometry object.**

**REQ. 83** **If the optional attribute metadata is not implemented or has a NULL value, the needed information shall be from this document.**

**REQ. 84** **If the attribute metadata is not NULL, then the first URI in the array shall be a normative document describing the standard implementation of this object.**

#### 6.4.4.19 Attribute representativePoint: DirectPosition

The attribute representativePoint is a coordinate location that is somewhere on the geometric object. The "representativePoint" may be implemented in different ways. It is a point value (DirectPosition) that is guaranteed to be on (interior to) this Geometry. The default logic may be to use the DirectPosition of the point returned by the operation "Geometry::centroid" if that point is interior to the object. Another use of representativePoint may be for the placement of labels in systems based on graphic presentation. Definitions for symbology and type placement are outside the scope of this document.

```
Geometry::representativePoint:DirectPosition
```

**REQ. 85** **The attribute representativePoint shall return a position interior to the geometric object**

#### 6.4.4.20 Attribute rsid: RSID [1..*]

The attribute "rsid" identifies the reference system used for all coordinates used within the geometry.

```
Geometry::rsid:RSID[1..*]
```

**REQ. 86** **All direct positions and geometry objects returned by accessing a geometric object shall use the reference system identified by the "rsid" unless another RSID is implied or supplied by the interface.**

In some geometry types, additional information associated with direct positions may be required to calculate interpolations (e.g. homogeneous weights for rational splines). These additional values may be stored at the "end" of the direct position's coordinate array.

**REQ. 87** **If the geometry object inherits a reference system from a container, then the local reference system shall be consistent with the inherited system in that the rsid identifier of the container shall be the initial part of the rsid of the geometric object.**

NOTE This is common for lists of parameters that can be extended in such a manner that new parameters are added on the "right" end of the parameter list.

#### 6.4.4.21 Attribute spatialDimension: Integer

The attribute spatialDimension is the number of spatial axis in the reference system. The normal values are 2 or 3.

```
Geometry::spatialDimension:Integer
```

#### 6.4.4.22 Attribute topologicalDimension: Integer

The attribute topologicalDimension is the largest topological dimension of the components of the object.

```
GeometrytopologicalDimension:Integer
```

For primitives, the topological dimensions are:

```
Empty  =  -1-dimensional
Point  =  0-dimensional
Curve  =  1-dimensional
Surface  =  2-dimensional
Solid  =  3-dimensional
```

**REQ. 88** **The dimension of any collection shall be the maximum of the dimensions of the contained primitives.**

#### 6.4.4.23 Attribute type: GeometryType [1..*]

The descriptions of geometric interpolations in the document are all defined in interfaces, not instantiable classes. The details of the instantiation of each type of interpolation may vary across implementations that are only obliged to follow the forms of the defined interfaces. The attribute type shall hold a reference back to the interface (or interfaces) from this document or its extensions that are supported by this particular instantiated object.

```
Geometry::type:GeometryType[1..*]
```

EXAMPLE A designed roadbed may change interpolation mechanism, and be presented as collections of reaches, road segments between intersections. Such a road may carry a type list including (curve, Geometry Complex).



**REQ. 89** **The type of the geometry object shall contain a value or values from the local GeometryType codelist which contains all instantiable types for geometry objects supported by the system.**

**REQ. 90** **The type value returned shall be the most specific types applicable to this particular instance of geometry object.**

```
Association Interior to:
Role containingGeometry: Geometry [0..*]
Role containedGeometry: Geometry [0..*]
```

In some cases, it may be difficult to determine if one geometry object is contained in another by a simple comparison of representation, usually when one of the geometry objects is a smaller dimension than that of the other. For example, a point on a surface may be subject to "round off" errors that may make a coordinate calculation ambiguous. The purpose of the association "Interior to" is to make this relation explicit in those cases subject to ambiguity.

#### 6.4.4.24 Operation buffer (radius: Distance): Geometry

**REQ. 91** **The operation "buffer" shall return a Geometry containing all direct positions with distance to this geometry of less than the given radius distance.**

```
Geometry::buffer(radius:Distance):Geometry
```

#### 6.4.4.25 Operation: dimension (point: DirectPosition = NULL): Integer

**REQ. 92** **The operation "dimension" shall return the inherent topological dimension of this Geometry, which shall be less than or equal to the coordinate dimension.**

The dimension of a collection of geometric objects shall be the largest dimension of any of its pieces. Points are 0-dimensional, curves are 1-dimensional, surfaces are 2-dimensional and solids are 3-dimensional. Locally, the dimension of a geometric object at a point is the dimension of a neighborhood of the point – that is the dimension of any coordinate neighborhood of the point. Dimension is unambiguously defined only for DirectPositions interior to this Geometry.

**REQ. 93** **If the passed DirectPosition for dimension is NULL, then the operation shall return the largest possible dimension for any DirectPosition in this Geometry's interior.**

```
Geometry::dimension(point:DirectPosition=NULL):Integer
Operation: distance (geometry: Geometry): Distance
Operation: distance (point: DirectPosition): Distance
```

#### 6.4.4.26 Operation: distance

**REQ. 94** **The operation "distance" shall return the distance between this Geometry and another Geometry, or position.**

The two required operations are:

```
distance(geometry:Geometry):Distance //to a geometry

distance(point:DirectPosition):Distance //to a point
```

"Distance" is one of the units of measure data types defined in ISO 19103.

The role of the reference system in distance calculations is important. There are three types of distances between points (and therefore between geometric objects): map distance, geodesic distance, and terrain distance. Map distance is the distance between the points as defined by their positions in a coordinate projection (such as on a map when scale is taken into account). Map distance is only accurate where the projection's point scale factor is close to unity, but map distance is also correctable for accuracy for small areas where scale functions have well-behaved derivatives that allow for local correction. Geodesic distance is the length of the shortest curve between those two points along the surface of the Earth model being used by the coordinate reference system. Geodesic distance behaves well for wide areas of coverage, and considers the Earth's curvature. It is especially handy for air and sea navigation, although care should be taken to distinguish between rhumb line (curves of constant bearing) and geodesic curve distance. Terrain distance takes into account the local vertical displacements (hypsography). Terrain distance can be based either on a geodesic distance or on a map distance. It is necessary to assure that geometry objects are topologically closed (by definition). Since the distinction between less than a distance or equal to a distance is beyond the capability of any digital computation, this should not matter since the inherent limitation of digital computations makes distinguishing between less than or equal to a distance. There are cases for which this Geometry would be partially outside the domain of validity of the object's reference system. If this case should arise, the implementation shall decide on appropriate action.

#### 6.4.4.27 Operation equal

The function "equal" for Geometry is defined by set equality (Geometry is defined as a transfinite set of direct positions). If A and B are instances of implementation subtypes of Geometry, then they are spatially equal if all spatial positions in A are also in B, and all spatial positions in B are also in A. If no coordinate system is passed, the assumed projection is the spatial coordinates. Therefore a spatiotemporal coordinate system for the geometry would only be checked for spatial equality.

Because of this, any empty geometry is equal to any other empty geometry.

If a GeometricCoordinateSystem is given, the set equality is based on the projection of the objects into the GeometricCoordinateSystem (6.4.4.20). For example, if a parameter "t" to represent time has been added, then the equality would be spatiotemporal.

**REQ. 95** **If the coordinate system of the passed object is changed to the coordinate system of this object, the operation "equal" shall return TRUE if and only if the converted Geometry is equal to this Geometry in its coordinate system.**

**REQ. 96** **The operation "equal" shall return TRUE if the passed Geometry when projected to spatial coordinates only, is equal to this Geometry if it were also projected to its spatial coordinates only.**

```
equal(geometry:Geometry):Boolean

equal(geometry:Geometry,cs:GeometricCoordinateSystem):Boolean
```

#### 6.4.4.28 Operation transform

The operation transform returns a new geometric object, in the new reference system indicated by the passed RSID, equal to this geometric object within the accuracy of the transformation.

**REQ. 97** **Each Geometry can produce a representation of itself in a given reference system supported by the implementation**

```
transform(new:RSID):Geometry
```

NOTE The behaviour of a transform in non-spatial dimensions can be problematic. Normally, non-spatial coordinates will have their own internal logic for transformations to other compatible systems. If the Geometry is purely spatial, then this transform is based on a change of coordinates from one SC_CRS to another.

### 6.4.4.29 Constructors Geometry

There are four constructors for geometry:

```
Geometry(data: GeometryData): Geometry
Geometry(text: CharacterString):Geometry
Geometry(binary: Binary):Geometry
Geometry(xml: XML):Geometry
```

**REQ. 98** **The constructor Geometry shall return a Geometry object of the class most consistent with the passed data.**

This operation will be inherited by all subtypes of geometry and will have to implement this for its own implementation class. It may rename the constructor locally to correspond to its class. For example:

```
Curve.Geomtry(CurveData) = Curve.Curve(CurveData)
Curve.Geometry(text:CharacterString):Geometry
Curve.Geometry(binary:Binary):Geometry
Curve.Geometry(xml:XML):Geometry
```

Each default datatype for geometry will always produce a version of itself regardless of which Geometry constructor gets the data. For example, if a generic Curve constructor gets a LineData input, its default action is to locate a "Line" implementation and pass the constructor datatype to it. So, if an implementation supports Line, BSpline and Geodesic, then if any constructor gets GeodesicData:

```
Geometry(GeodesicData) = Line(GeodesicData) =
BSpline((GeodesicData) = Geodesic(GeodesicData)
```

**REQ. 99** **If any implementation of the Geometry constructor is passed a datatype that it cannot handle, it shall delegate the construction to the root Geometry constructor that in turn shall have access to a constructor for any subtype of Geometry supported by the system.**

Table below summarizes the data structures for the default constructors.

NOTE 1 Inheritance will repeat parameters for every subtyping relationship. Therefore, for example, the RSID in GeometryData is inherited by all constructors of geometry interfaces. The "interpolation" codelist values are not used, since the name of the constructor type implies the interpolation.

EXAMPLE 1 The default constructor data for line, rhumb and geodesic are identical since only the "interpolation" differs between them.

If an implementation decides on a generic free-function to construct geometry instances, then the geometry type would have to be included as a first parameter. In encoding this would be reflected in the record type.

For orientable primitives, if the orientation is positive "+", it may be omitted from the base constructor. If the orientation is negative "–", then both a positive and negative instantiation of the primitive should be constructed. If the encoding mechanism has an identity and reference mechanism, then the negative primitive should contain a reference to the positive instance. This would support the constraints on Orientable.

EXAMPLE 2 Following the pattern below, the constructor list for "Line" would be an RSID from Geometry, a dataPoint list and knot list from Curve and a type label "LINE". Therefore the BNF for a line geometry would follow the subtyping tree and would look something like this:

```
LINE(<RSID>,<DATAPOINT LIST>,<KNOT LIST>) =
LINE0(<Geometry DATA LIST>,<ORIENTABLE LIST>,
<CURVE DATA LIST>,<LINE DATA LIST>)
```

The <LINE DATA LIST> would be empty, from the table below.

The GeometryData datatype contains a derived parameter "type" specifying the type of geometry to be constructed. In an object system with class structures, this is not needed since the constructor called is associated to the type to be constructed. In a "free function" world (i.e. not classical OOPL), the type should be included in the constructor, since the "free function" may not be associated to a particular type. In well known text, the name of the type is usually the opening label. Another way to get the type is to use a "typeOf" call on the parameter, and drop the "Data" at the end. The inclusion in the model is for languages without a type system, such as JavaScript.

**Table 1 — Geometry Default Constructor Data Types**

| Default Constructor of | Supertype | Parameter Types (added to inheritance) |
|---|---|---|
| Geometry | | type:GeometryType, rsid:RSID[] |
| Collection | Geometry | element:Geometry[] |
| Product Curve | Collection, Curve | element:Curve[], projections:Projection [] |
| Point | Geometry | position:DirectPosition |
| Orientable (Primitive) | Geometry | orientation:sign |
| Curve | Orientable | dataPoint:DirectPosition[], knot:Real[] |
| Line | Curve | |
| Geodesic | Curve | |
| Rhumb | Curve | |
| Polynomial Arc | Curve | coordinateOffset:RealPolynomial[] |
| Polynomial Curve | Curve | degree:Integer, numArc:Integer, segment:PolynomialArc[] |
| Offset Curve | Curve | base:Curve, bearing: Bearing |
| Conic | Curve | centre:Position, angle:Bearing[], rotation:Rotation, isCycle:Boolean |
| Circle | Conic | radius:Distance |
| Spiral | Curve | curvature:RealFunction,torsion:RealFunction, startPoint:DirectPosition, startFrame:Vector[] |
| Bspline | Curve | isRational:Boolean,controlPoint:DirectPosition[] |
| Surface | Orientable | boundary:Curve[] |

**Table 1** *(continued)*

| Default Constructor of | Supertype | Parameter Types (added to inheritance) |
|---|---|---|
| Gridded Surface | Surface | dataPoint:DirectPosition[][] |
| Polyhedral Surface | Surface, Collection | |
| Polygon | Surface | |
| Solid | Geometry | boundary:Surface[] |

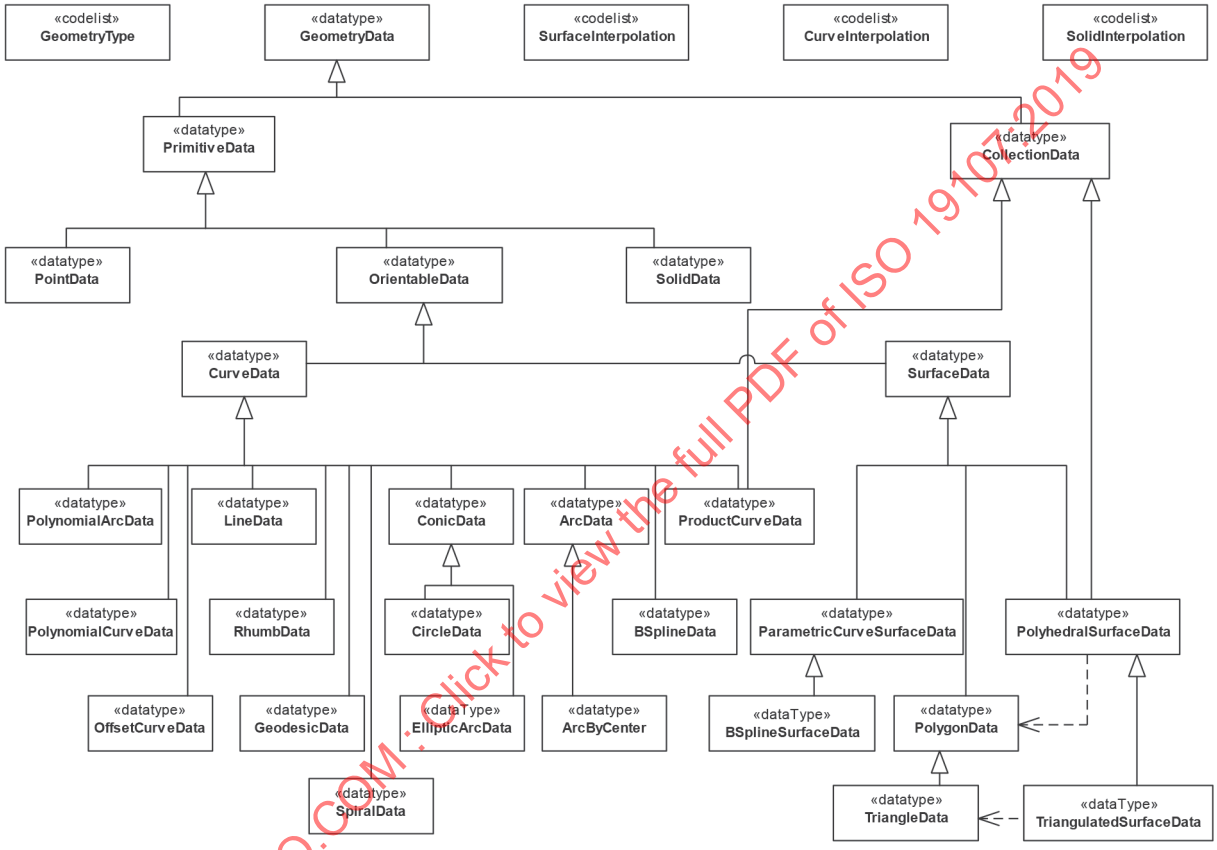



**Figure 11 — Geometry Default Constructor Data Types**



NOTE 2 In the above table, several default constructor datatypes are empty. In these cases, the contents of the default-constructor do not vary from the inherited datatype from its superclasses. For example, LineData can use the same structure as CurveData, but it will use a "linear" interpolation type, which is constant for the Line constructor operation. This is also true for Rhumb and Geodesic curves. All use the same constructor values but use different interpolation because of their class.

### 6.4.4.30 Operations from TransfiniteSet realization

Each Geometry represents a potentially infinite set of points in its reference systems. This means that a Geometry acts as a generalized set in some calculations, and is a subtype TransfiniteSet of DirectPosition in the "coordinateSystem" of the Geometry.

**REQ. 100 All Geometry instances shall support all common set theory operations.**

The Geometry instances realize the following operations from the Interface TransfiniteSetOfDirectPositions.

```
element:DirectPosition):Boolean                          //element of
contains(set:Geometry):Boolean                           //subset
intersects(set:Geometry):Boolean                         //intersects
equals(set:Geometry):Boolean                             //equals A=B?
union(set:Geometry):Geometry                             //union, A∪B
intersection(set:Geometry):Geometry                      //intersection
difference(set:Geometry):Geometry                        //A-B
symmetricDifference(set:Geometry):Geometry               //(A-B)∪(B-A)
```

The "symmetricDifference" operation returns the set symmetric difference of this Geometry and the passed Geometry. The set definition of symmetric difference or "exclusive or" difference ($A \oplus B$, A + $B$, $A \Delta B$, or $A \nabla B$ ; $A \oplus B$ preferred) is:

$$\boldsymbol{A}, \boldsymbol{B} \in \boldsymbol{Set} \Rightarrow \left[ \boldsymbol{A} \oplus \boldsymbol{B} = (\boldsymbol{A} - \boldsymbol{B}) \cup (\boldsymbol{A} - \boldsymbol{B}) = (\boldsymbol{A} \cup \boldsymbol{B}) - (\boldsymbol{A} \cap \boldsymbol{B}) \right] \tag{16}$$

### 6.4.5 Datatype GeometryData

The default constructor datatype for Geometry is GeometryData. It is the root of the datatype hierarchy for the classes under the Geometry interface. It will contain parallel attributes for the "rsid" attribute in Geometry (0) and the type attribute (0).

### 6.4.6 CodeList: GeometryType

The codelist GeometryType is a local list of types inheriting from Geometry. It essentially represents the software contract between the implementation and its users for the support of types of geometry, based on dimension and mechanism of interpolation.

**REQ. 101** **All subtypes of Geometry supported by an implementation shall be listed in the local copy of the GeometryType codelist. This list shall always include the root interface geometry, empty and point.**

### 6.4.7 Interface Encoding

The Encoding sub-interface for Geometry supplies a common set of operations for moving between text or binary representations of Geometry and object instantiations. The three common alternative representations are:

— GML (any version may be supported)

— WKT (well-known text)

— WKB (well-known binary)

**REQ. 102** **Each geometry subtype shall support attributes and constructors that support well-known text (WKT), Geography Markup Language (GML), well-known binary (WKB) and the datatype GeometryData including its subtypes, allowing any instance to return values for these formats and be constructed from valid instances of elements given in these formats.**

The required attributes are:

```
asText:CharacterString          //WKT character string
asBinary:Binary                 //WKB Bit string
asGML:CharacterString           //GML character string (XML fragment)
asData:GeometryData             //associated class datatype
```

The required constructors are:

```
Geometry(text:CharacterString):Geometry                 //from WKT
Geometry(bin:Binary):Geometry                           //from WKB
Geometry(gmlText:CharacterString):Geometry              //from GML
Geometry(data:GeometryData)Geometry                     //from datatype
```

### 6.4.8 Interface Query2D

#### 6.4.8.1 Semantics

The Query2D sub interface of Geometry supplies interfaces for query in 2D coordinate spaces or in 3D coordinate spaces where the geometry objects are projected onto the horizontal surface (the GeometricReferenceSurface in use). For each of the operations in this interface, each of the geometric objects (if not already 2D) is projected onto the Reference Surface being used in the geometric coordinate system.

#### 6.4.8.2 Operation: distance

The operation distance returns the distance between the input geometric object and this geometric object after they have been projected onto the GeometricReferenceSurface being used.

```
distance (another: Geometry): Distance
```

This distance is the minimal distance between two points in the two projected objects. If the objects overlap, then the distance is zero.

This distance is defined to be the greatest lower bound of the set of distances between all pairs of points that include one each from each of the two geometry objects. If necessary, the second geometric object shall be transformed into the same coordinate system as the first before the distance is calculated.

The return value may be in one of the units of measure datatypes defined ISO 19103.

#### 6.4.8.3 Operation: buffer (distance: Distance): Geometry

The buffer operation creates a new geometry object that contains all direct positions that are within a distance (less than or equal to the distance) of this geometry object.

```
buffer (distance: Distance): Geometry
```

NOTE It is necessary to assure that geometry objects are topologically closed (by definition). Since the distinction between less than a distance or equal to a distance is beyond the capability of any digital computation, this should not matter since the inherent limitation of digital computations makes distinguishing between less than or equal to a distance.

The distance operation returns the distance between the parameter geometry object and this geometry object. The distance will be consistent with the GeometricReferenceSurface in use, and the distance on the Reference Surface as is consistent with the 2D nature of this interface (Query2D). Thus, the operation buffer creates a new geometric object containing all the direct positions whose distance from this geometric object is less than the radius passed. If the radius is "0", the resulting object is equal to this geometric object. If the radius is negative, the returned geometric object contains all directions

positions within this object whose distance from the boundary is greater than the absolute value of the passed radius. The Geometry returned is in the same reference system as this original Geometry. The topological dimension of the returned Geometry is normally the same as the spatial dimension of the reference system  a collection of Surfaces in 2D space and a collection of Solids in 3D space, but this may be application defined.

```
buffer(distance:Distance):Geometry
```

There are cases for which this Geometry would be partially outside the domain of validity of the object's reference system. If this case should arise, the implementation shall decide on appropriate action.

#### 6.4.8.4 Set operation: intersection

The operation intersection returns the best geometry object that corresponds to the set theoretic intersection of the parameter geometry object and this geometry object. Since the intersection of two closed sets is closed, the set theoretic answer should be equal to the topological answer (within the limitation of the metric accuracy of the geometric representations).

```
intersection(another:Geometry):Geometry
```

#### 6.4.8.5 Set operation: difference

The operation difference returns the best geometry object that corresponds to the set theoretic difference of this geometry object and the parameter geometry object. Since the difference of two closed sets is not necessary closed, the set theoretic answer may differ from the topological answer along its boundary (within the limitation of the metric accuracy of the geometric representations).

```
difference(another:Geometry):Geometry
```

#### 6.4.8.6 Set operation: symDifference

The operation symmetric difference (symDifference) returns the best geometry object that corresponds to the set theoretic symmetric difference of the parameter geometry object and this geometry object. Since the symmetric difference of two closed sets is not necessary closed, the set theoretic answer may differ from the topological answer along its boundary (within the limitation of the metric accuracy of the geometric representations).

```
symDifference(another:Geometry):Geometry
```

#### 6.4.8.7 Set operation: union

The operation union returns the best geometry object that corresponds to the set theoretic union of the parameter geometry object and this geometry object. Since the union of two closed sets is closed, the set theoretic answer should be equal to the topological answer (within the limitation of the metric accuracy of the geometric representations).

```
union(another:Geometry):Geometry
```

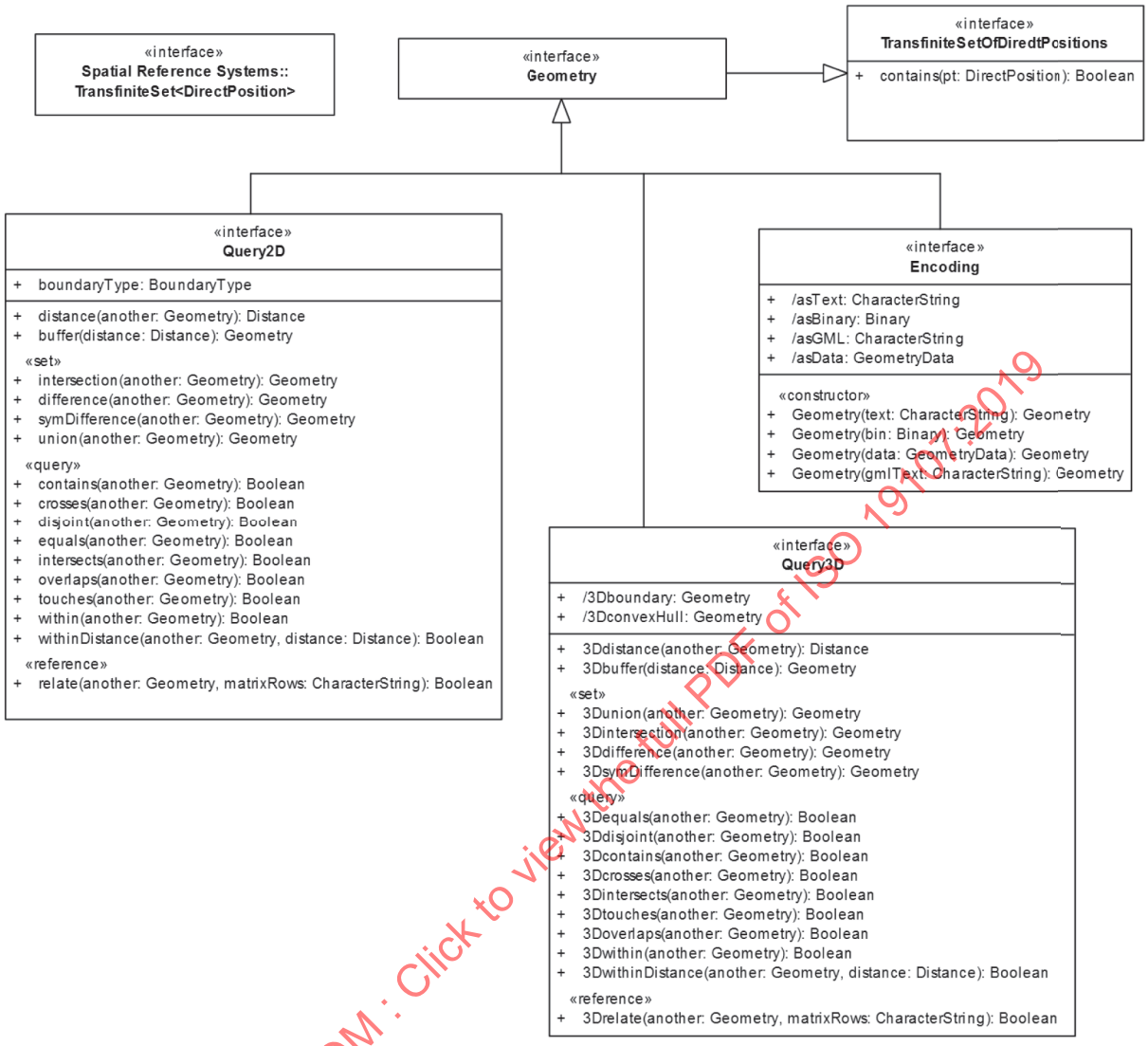



**Figure 12 — Geometry Root Extensions**



### 6.4.8.8 Map query operations

The topological query operations are generally dependent of the work following from [33], which allows them to be defined by a 2 by 2 matrix, but later under a 3 by 3 matrix or possible outcomes of set intersections. This is discussed fully in 10.8.

```
contains(another:Geometry):Boolean                  // A contains B

crosses(another:Geometry):Boolean                   // A crosses B

disjoint(another:Geometry):Boolean                  // A disjoint from B

equals(another:Geometry):Boolean                    // A equals B

intersects(another:Geometry):Boolean                // A intersects B

overlaps(another:Geometry):Boolean                  // A overlaps B

touches(another:Geometry):Boolean                   // A touches B

within(another:Geometry):Boolean                    // A interiour to B
```

```
withinDistance(another:Geometry,distance:Distance):Boolean
                                                    // A within distance to B
relate(another:Geometry,matrix:CharacterString):Boolean Boolean
                                                    // Referemce using matrix
```

Some of these operations are equivalent to simple set operations as follows:

A.contains(B)⟺A⊇B  A.disjoint(B)⟺A⋂B=∅

A.equals(B)⟺A=B  A.intersets(B) ≠∅

A.within(B)⟺B.contains(A)

### 6.4.9 Interface Query3D

The Query3D sub-interface of Geometry supplies interfaces for query in 3D coordinate spaces. Since these operations are only distinct from those in Query2D if the objects and all parameters of type Geometry in the operation are also 3D spatial, all object supporting Query3D are 3D spatial objects.

**REQ. 103 All Geometry objects in a Query3D operation are also of type Query3D**

**REQ. 104 For instances of Query3D, "is3D" shall always be TRUE.**

The specific 3D attributes are:

```
3Dboundary:Geometry          // a spatial 3D boundary
3DconvexHull:Geometry        // a spatial 3D convex hull
```

The specific 3D operations are:

```
3Ddistance (another:Geometry):          Distance   // 3D distance
3Dbuffer (distance:Distance):            Geometry   // 3D buffer
```

The set operations and relations are also special for 3D and are as follows:

```
3Dintersection(another:Geometry):Geometry            //A intersect B
3Ddifference(another:Geometry):Geometry              //A-B
3DsymDifference(another:Geometry):Geometry           //(A-B)U(B-A)
3Dunion(another:Geometry):Geometry                   //AUB
3Dcontains(another:Geometry):Boolean                 //A⊆B
3Dcrosses(another:Geometry):Boolean                  //A⊇B
3Ddisjoint(another:Geometry):Boolean                 //A⋂B=∅
3Dequals(another:Geometry):Boolean                   //A=B
3Dintersects(another:Geometry):Boolean               //A⋂B≠∅
3Doverlaps(another:Geometry):Boolean                 //see Clause 10.8
3Dtouches(another:Geometry):Boolean                  //see Clause 10.8
3Dwithin(another:Geometry):Boolean                   //see Clause 10.8
```

```
3DwithinDistance(another:Geometry,dist:Distance):Boolean

            //A.distance(B) is less than "dist"

3Drelate(another:Geometry,matrix:CharacterString):Boolean

            //reference implementation of topological operators (10.8)
```

Some of these operations are equivalent to simple set operations as follows:

`A.3Dcontains(B)` $\Leftrightarrow A \supseteq B$

`A.3Ddisjoint(B)` $\Leftrightarrow A \cap B = \emptyset$

`A.3Dequals(B)` $\Leftrightarrow A = B$

`A.3Dintersets(B)` $\neq \emptyset$

`A.3Dwithin(B)` $\Leftrightarrow$ `B.contains(A)`

### 6.4.10 Interface Empty

The implementation of the isEmpty attribute at Geometry (see 6.4.4.2) may require the existence of an Empty object class. Whether a geometry object is empty because the isEmpty is set to TRUE or because it was created as an instance of the class EMPTY, its behaviour is the same and describe here and in 6.4.4.2. In either case, the implementation of this behaviour is an implementation prerogative.

The empty set is unique, but it may be represented by any subtype of Geometry through the "isEmpty" Boolean attribute. There is only one empty set, because the set theoretic definition of equality implies that any ∅ is always equal to any other ∅.

$$\boldsymbol{A,B} \in \boldsymbol{Set}: \left[\boldsymbol{A} = \boldsymbol{B}\right] \Leftrightarrow \left[\left[\boldsymbol{x} \in \boldsymbol{A}\right] \Leftrightarrow \left[\boldsymbol{x} \in \boldsymbol{B}\right]\right] \tag{17}$$

It is required in this document to allow the set operations (see 6.4.4.29) to have a proper return type where empty results are possible (e.g., the intersection of two disjoint geometries or the difference of any geometry and itself). Since the Empty set does not have a defined dimension, no defined coordinate system, and nothing that distinguishes it from any other ∅, it does not properly belong to any of the primitive classes, but can use the "isEmpty" Boolean to be represented by instances of any geometry primitive class.

Math Some of the behavior of the empty geometry (the empty set, ∅), especially when it interact with itself geometrically, is possibly counter-intuitive, so the following facts are given (all mathematically provable from the definitions):

The distance from the empty set, ∅, to any other non-empty geometry is +∞. This is from the use of glb (greatest lower bound) or min in the definition of distance:

$$\text{distance}(A,B) = \min\{c.length \mid c \in Curve, c.startPoint \in A, c.endPoint \in B\} \tag{18}$$

If either A or B is ∅, then their distance apart is infinite (+∞).

$$\left[A = \emptyset \vee B = \emptyset\right] \Rightarrow \left[A.\text{distance}(B) = \min(\emptyset) = +\infty\right] \tag{19}$$

For any geometry for which isEmpty=TRUE, then the following hold:

— A buffer, convex hull, boundary, canonical representation or envelop of ∅ is ∅.

— The topological dimension of empty is minus one (*dim* (∅) = –1).

— The Booleans isEmpty, isCycle, isSimple and isValid are TRUE for ∅.

— The centroid and representativePoint of ∅ are both ∅.

— The return values of stroke, transform, mapProject (or any projection) of ∅ is ∅.

— Any intersection with ∅ is ∅ $\left(A \cap \varnothing = \varnothing\right)$.

— Any union of ∅ and A is A $\left(A \cup \varnothing = A\right)$.

— The difference and symmetric difference of A and ∅ is A ($A - \varnothing = A$).

— The difference of ∅ and anything is ∅ ($\varnothing - \boldsymbol{A} = \varnothing$).

— The distance from ∅ to ∅ or any other geometry is +∞.

— ∅ is a subset of every set; ∅ is a superset only of itself.

The set "{∅}" (the set of sets containing only the empty set) and ∅ are not the same. The cardinality of ∅ is zero. The cardinality of {∅} is one.

### 6.4.11 Interface Primitive

#### 6.4.11.1 Semantics

For the purposes of this document, a geometry primitive is a connected geometry object with a uniform dimension at every interior point. Depending on the spatial dimension of the coordinate space, the primitives consist of subclasses of Point, Curve, Surface and Solid.

| | |
|---|---|
| –1 | Empty set, containing nothing |
| 0 | Point, 0-dimensional "positions"; each point is isolated. |
| 1 | Curves, 1-dimensional geometry objects where each non-boundary point is embedded in a local neighborhood topologically isomorphic to an open interval on the "number line". |
| 2 | Surfaces, 2-dimensional geometry objects where each non-boundary point is embedded in a local neighborhood topologically isomorphic to the interior of the unit circle in the Cartesian 2D space, i.e. a plane. |
| 3 | Solids, 3-dimensional geometry objects where each non-boundary point is embedded in a local neighborhood topologically isomorphic to the interior of the unit sphere in a Cartesian 3D space. |

A large number of the geometric types in this document are defined parametrically, that is they are represented by functions from a set of parameters (in a parametric space, usually a subset of some Euclidean n dimensional coordinate space or $\mathbb{E}^n$) into a coordinate space of some equal or larger dimension. The first few dimensions (up to 3) representing geographic space, the next possibly time, and any remainder representing whatever the application needs, such as distributed attributes or some other measures. The type of geometry is usually determined by the dimension of the parameter space, which will normally be equal to the topological dimension of the resulting geometry. Therefore, a 0-parameter geometric object is a point, 1-parameter geometric object is a curve, a 2-parameter geometric object is a surface and a 3-parameter geometric object is a solid. To maintain this, a parametric mapping defining a geometric object should be locally bicontinuous to maintain topological dimension. This means in the parameter space, around any parametric point is a possibly small neighbourhood in which the spatial projection of the parametric mapping is one-to-one with a continuous inverse. This would mean that locally, the geometric object would be an image of n dimensional Euclidean space, which forces its topological dimension to be the same as the number of parameters.

In each part of the included model, this version of this document includes generalizations of the standard geometric objects (points, curves, surfaces and solids) to more complex and flexible interpolation algorithms.

The Geometric primitive package contains all the root geometric primitives. Primitives are considered the minimal pieces of geometry to be dealt within the context of values. Since most sets of points in



geometry are infinite (except for points), primitives can usually be split into smaller pieces that would also represent geometric primitives, so the usual definition of primitives being non-decomposable cannot be used. In geometry, primitives are objects that are chosen not to be decomposable into finer geometric objects. They can be decomposed into segments or patches (which are of the same dimension) that are used for representational purposes (such as interpolation). The closest semantic usage here is classical Euclidean geometry, where lines, circles and other geometric constructs are treated as a single entity. Operation may further break them apart as smaller geometric entities, but the originals still maintain their identity, and structure.

Implementations wishing to combine the functions of geometric primitive types and interpolating structures should use the multiple realizations of these sets of interfaces in defining classes within their application or profile schema. Care should be taken since interface realization carries the intent of substitutability.

Geometry Primitive (Figure 13) is the abstract root class of the geometric primitives. Its main purpose is to define the basic "boundary" operation that ties the primitives in each dimension together. A geometric primitive is a geometric object that is not decomposed further into other primitives in the system. This includes curves and surfaces, even though they are composed of curve and surface segments. This composition is a strong aggregation: curve and surface segments cannot exist outside the context of a primitive.

NOTE Most geometric primitives are decomposable infinitely many times. Adding a centre point to a line can split that line into two separate lines. A new curve drawn across a surface can divide that surface into two parts, each of which is a surface. This is the reason that the normal definition of primitive as "non-decomposable" is not plausible in a geometry model – the only non-decomposable object in geometry is a point. Any geometric object that is used to describe a feature is a collection of geometric primitives. A collection of geometric primitives may or may not be a geometric complex. Geometric complexes have additional properties such as closure by boundary operations and mutually exclusive component parts.

#### 6.4.11.2 Association: Role segment

The role "segment" lists the components (contained smaller primitives of the same dimension) of Primitive, each of which defines a portion of the Primitive. The order of the segments is the order in which they are used to define the Primitive.

```
Primitve::segment: Primitve [0..*]
```

**REQ. 105** **Points shall have no segments.**

**REQ. 106** **Curves may have segments; the order shall imply the order of the curves in the relation role.**

**REQ. 107** **The curve shall be traced first in the composite, then each segment shall be traced recursively in its order in the segment array.**

**REQ. 108** **In the case of Surfaces or Solids, the order may be ignored.**

In Curves, it indicates the order for tracing. Since the composition is recursive, the traversal is specified to be depth first. First the root curve is traversed, and then each of its segment curves are traversed in the order of the role. The various parameter intervals are shifted as needed to make the totality of the parameter spaces continuous.

#### 6.4.12 Datatype PrimitiveData

The datatype, in addition to the inherited attribute from GeometryData, will have an attribute "segment" of type "PrimitiveData*[0..]" referencing PrimitveData instances representing the associated segments defined in Clause 0 "segment"

### 6.4.13 Interface Point

#### 6.4.13.1 Semantics

A Point instance of the Geometry is a single location given by a direct position (Figure 14).

#### 6.4.13.2 Attribute position: DirectPosition

The attribute "position" gives the location of the Point in its reference system. The distinction between Point and DirectPosition is that Point as an object instance has a system supplied identity, but an instance of DirectPosition is a data type whose only identity is its value.

```
Point::position:DirectPosition
```

#### 6.4.13.3 Attribute boundary: Geometry = Empty

The attribute "boundary" gives the boundary of the Point as a Geometry reference system. Since the boundary of a point is always empty, the boundary object will always have isEmpty=TRUE.

```
Point::boundary:Geometry.isEmpty
```

**REQ. 109** **For all Points P, P.boundary.isEmpty shall always be TRUE and P.segment→length = 0.**

$$p \in \text{Point} \Rightarrow [\text{p.boundary.isEmpty}] \wedge [\text{p.segment} \rightarrow \text{length}=0] \tag{20}$$

This is the reason that dimension of ∅ is considered to be −1. In most cases, where $\partial$ is the boundary operator, the following is true:

$$\dim(A) = \dim(\partial A) + 1 \tag{21}$$

If a point is dimension 0, this would imply that the dimension of ∅ would have to be −1.

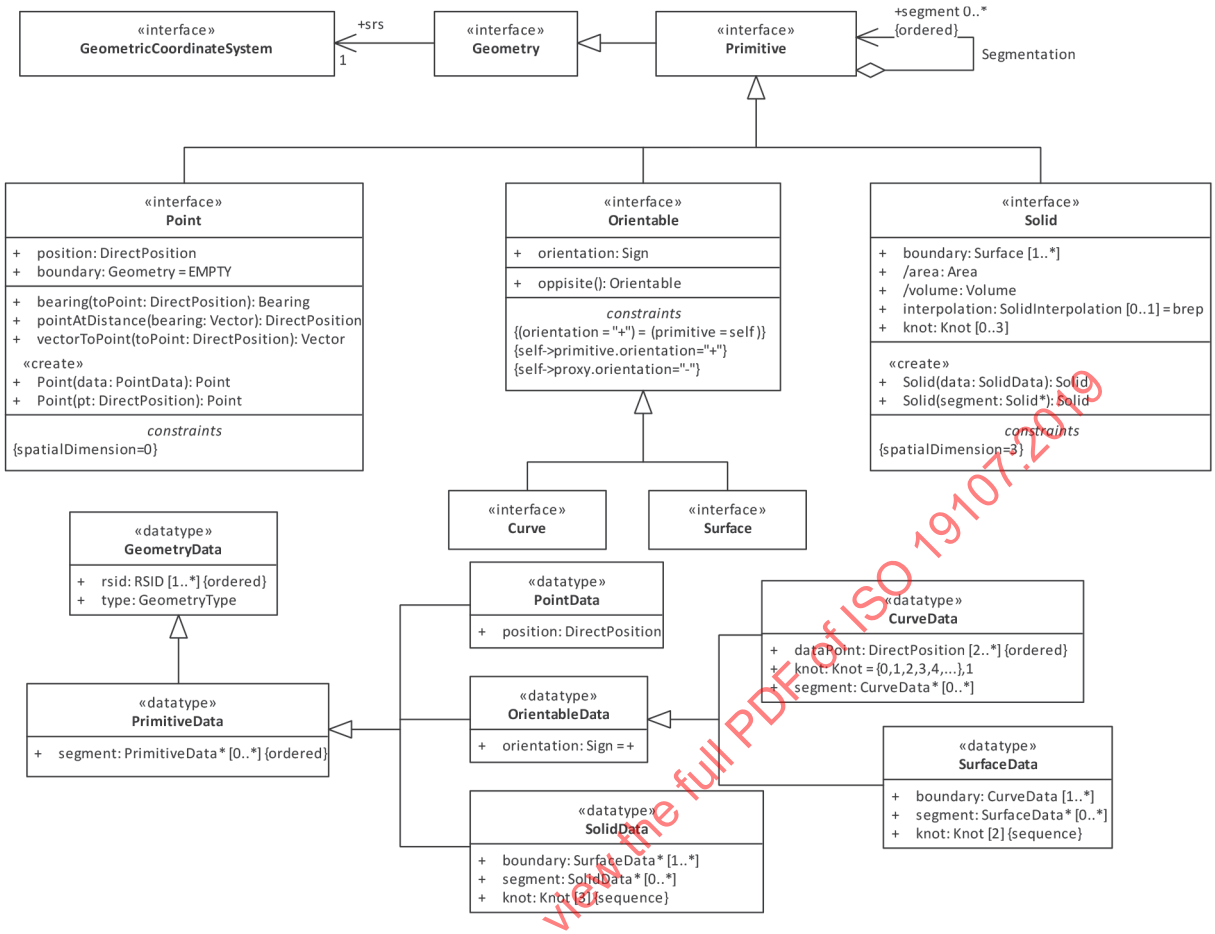



**Figure 13 — Geometry Primitives and Their Default Constructors**

### 6.4.13.4 Operation: vectorToPoint (toPoint: DirectPosition): Vector

The operation "vectorToPoint" will return a vector in the tangent space at the point whose direction determines a geodesic curve that intersects "toPoint" DirectPosition at a distance equal to the length of the vector. This operation solves the second geodetic problem.

```
Point::vectorToPoint(toPoint:DirectPosition):Vector
```

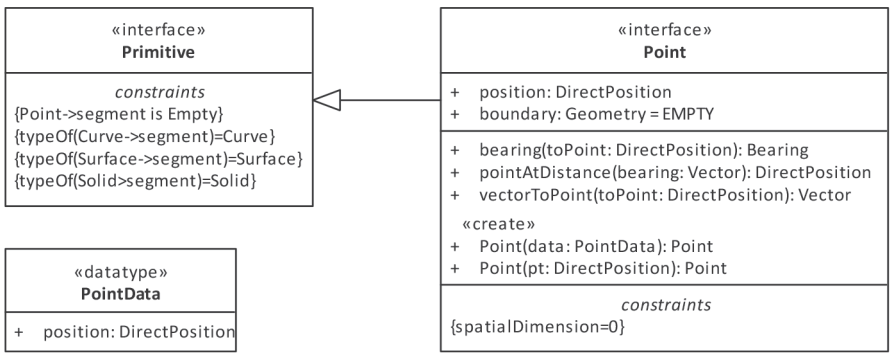



**Figure 14 — Point**

#### 6.4.13.5 Operation: Bearing (toPoint: DirectPosition): Bearing

The operation "Bearing" is similar to "vectorToPoint" without the distance information in the vector. It is essentially a constructor for Bearing based on this point and a target point.

```
Point::Bearing(toPoint:DirectPosition):Bearing
```

#### 6.4.13.6 Operation: pointAtDistance (bearing: Vector): DirectPosition

The operation "pointAtDistance" will return a DirectPosition given a vector in the tangent space at the point whose direction determines a geodesic curve that intersects that DirectPosition at a distance equal to the length of the vector. This operation solves the first geodetic problem.

```
Point::pointAtDistance(bearing:Vector):DirectPosition
```

#### 6.4.13.7 Constructor: Point (pt: DirectPosition): Point Constructor: Point (data: PointData): Point

The constructor Point (overloaded) will create an instance of a Point geometry object, at the given direct Position.

```
Point::Point(pt:DirectPosition):Point

Point::Point(data:PointData):Point
```

### 6.4.14 Datatype PointData

The datatype PointData, subtyped under PrimitveData, will contain in addition to its inherited attributes, a DirectPosition in an attribute called position (0).

### 6.4.15 Interface Orientable

#### 6.4.15.1 Semantics

The interface Orientable supplies functionality for primitives that can be meaningfully reversed without changing their "set theoretic" definition.

Orientable primitives (Figure 15) are those that can be mirrored into new geometric objects in terms of their coordinate systems. For curves, the orientation reflects the direction in which the curve is traversed, that is, the sense of its parameterization. When used as boundary curves, the surface being bounded is to the "left" of the oriented curve. For surfaces, the orientation reflects which direction is "up", the "top" of the surface being the direction of a completing "z-axis" that would be the local upward. When used as a boundary surface, the bounded solid is "below" the surface. The orientation of points and solids has no immediate geometric interpretation in 3-dimensional space.

Orientable objects are essentially references to geometric primitives that carry an "orientation" reversal flag (either "+" = "same orientation") or "–" = "reversed orientation") that determines whether this primitive agrees or disagrees with the orientation of the referenced object.

NOTE There are several reasons for subclassing the "positive" primitives under the orientable primitives. First is a matter of the semantics of subclassing. Subclassing is assumed a "is type of" hierarchy. In the view used, the "positive" primitive is simply the orientable one with the positive orientation. If the opposite view were taken, and orientable primitives were subclassed under the "positive" primitive, then by subclassing logic, the "negative" primitive would have to hold the same sort of geometric description that the "positive" primitive does. The only viable solution would be to separate "negative" primitives under the geometric root as being some sort of reference to their opposite. This adds a great deal of complexity to the subclassing tree. To minimize the number of objects and to bypass this logical complexity, positively oriented primitives are self referential (are instances of the corresponding primitive subtype) while negatively oriented primitives are not. Orientable primitives are often denoted by a sign (for the orientation) and a base geometry (curve or surface). The sign datatype is defined in ISO 19103. If "c" is a curve, then "< +, c>" is its positive orientable curve and "< , c>" is its negative orientable curve. In most cases, leaving out the syntax for record "<, >" does not lead to confusion, so "<+, c>" may be written as "+c" or simply "c", and "< , c>" as " c". Curve space arithmetic can be performed if the curves align properly, so that:

```
For c, d:OrientableCurves
such that c.endPoint=d.startPoint
then (c+d)== <c,d>
```

### 6.4.15.2 Attribute orientation: sign

The "orientation" of an orientable primitive determines which of the two possible orientations this object represents. Changing the orientation of a curve would reverse its direction and swap the start and end of the curve. Changing the orientation of a surface would reverse the direction of the "up vector" and the direction of each of its boundary curves.

Each instance of an orientable primitive may proxy for itself and for its reverse. The attribute sign distinguishes these two cases, where "+" implies positive orientation, i.e. a proxy for itself, and "–" implies a proxy for its reverse. In 2D and 3D systems, the primitive can be either a curve or a surface. The reverse of a curve swaps start and end and thus determines the direction of parameterization. In a surface, the orientation is the direction of the top of the surface, and a negative "–" reverses top and bottom, in effect reversing the upward normal of the surface.

```
Orientable::orientation:Sign
```

### 6.4.15.3 Association Roles: proxy and primitive operation reverse (): Orientable

An orientable primitive (either a curve or a surface) will be associated with a proxy that is spatially equal with the opposite orientation. The proxy need not carry any information except the identity or its primitive.

The association role "proxy" will always be the proxy version of the primitive. The association role primitive will always be original primitive.

```
Orientable::proxy: Orientable

Orientable::primitive: Primitive
```

The operation "reverse()" will always return the opposite object in the pair.

### 6.4.16 Datatype OrientableData

The datatype OrientableData (Figure 15), subtyped under PrimitveData will contain in addition to its inherited attributes, a sign in an attribute called orientation (6.4.15.2).

### 6.4.17 Datatype Knot

#### 6.4.17.1 Semantics

The knots are values from the domain of a constructive parameter space for curves, surfaces and solids. Each knot sequence is used for a dimension of the parameter space $\overrightarrow{k_i} \in \{u_0, u_1, u_2 \ldots\}$. Thus, in a surface using a functional interpolation such as a b-spline, there will be two knot-sequences, one for each parameter, $\overrightarrow{k_{i,j}} = (u_i, v_j)$. For such an interpolated solid, there would be 3, $\overrightarrow{k_{i,j.k}} = (u_i, v_j, w_k)$. The sequence of knots act as fence points with the intervals lying between the post, so the $i^{th}$ interval is $[u_{i-1}, u_i]$.

In the knot sequence for a b spline, a knot can be repeated (affecting the underlying spline formulae). In other curves, knots will all be multiplicity 1. The number of repetitions is the multiplicity of the knot. The internal representation of the knot sequence has two equivalent representations. The first is a simple sequence, with repetitions of each knot for its multiplicity:

$$T = \{t_0, t_1, t_2, t_3, \&, t_n\} \text{ with } t_i \leq t_{i+1} \tag{22}$$

The second form is each knot (a real, in $\mathbb{R}$) is distinct and accompanied by a multiplicity (a positive integer, in $\mathbb{Z}$).

$$\boldsymbol{k_i} = (\boldsymbol{t_i} \in \mathbb{R}, \boldsymbol{m_i} \in \mathbb{Z}) \tag{23}$$

Either storage format is acceptable, but the first is more commonly used in the defining formulae for b-splines.

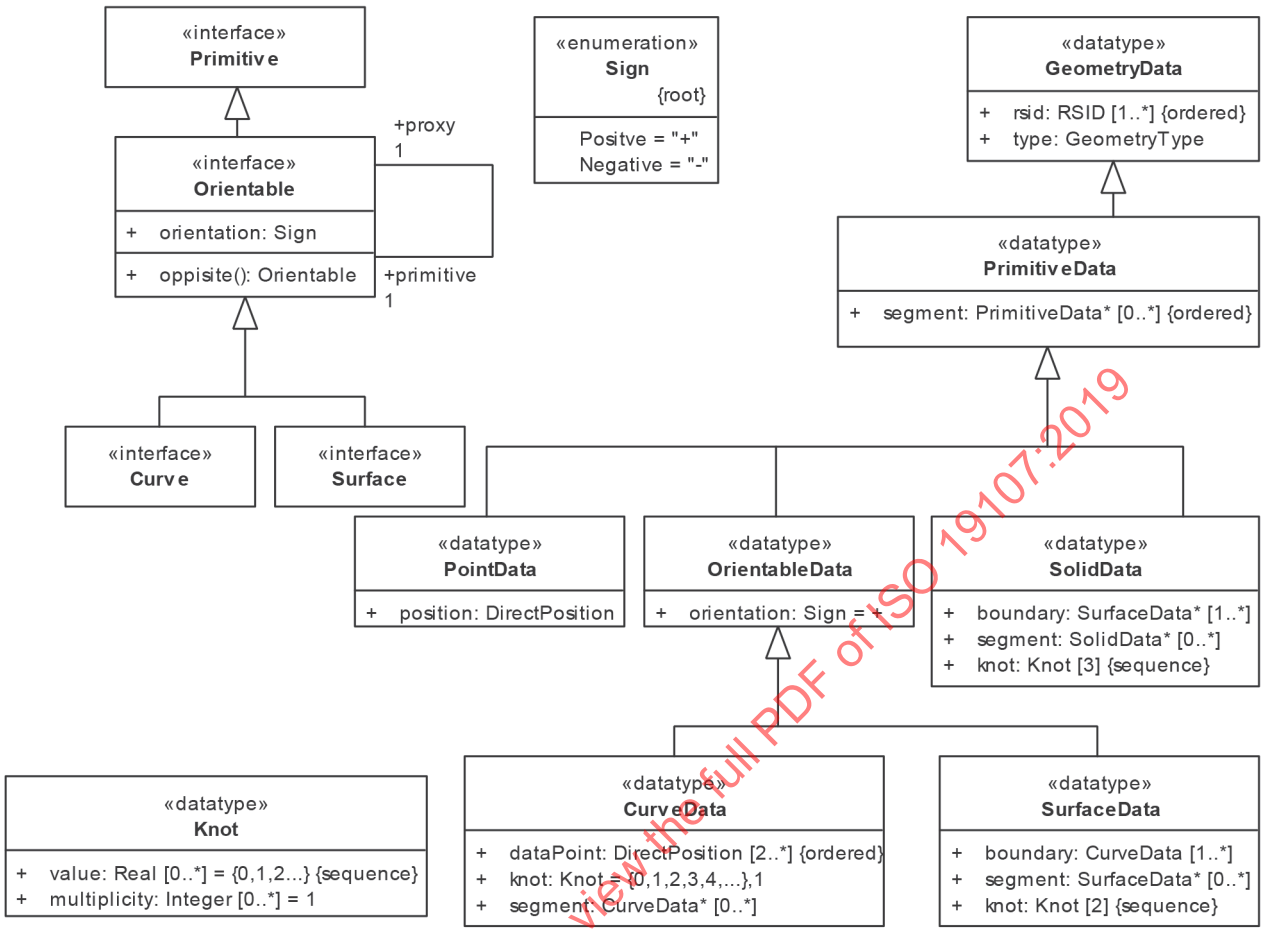



**Figure 15 — Orientable Primitive**



#### 6.4.17.2 Attribute value: Real

The attribute "value" is the value of the parameter at the knot of the spline. The sequence of knots shall be a non-decreasing sequence. That is, each knot's value in the sequence shall be equal to or greater than the previous knot's value. The use of equal consecutive knots is normally handled using the multiplicity.

```
Knot::value: Real[*]
```

#### 6.4.17.3 Attribute multiplicity: Integer

The attribute "multiplicity" is the multiplicity of each knot used in the definition of the curve. The semantics of the multiplicity is explained in the description of the various splines.

```
Knot::multiplicity: Integer[0..*]
```

### 6.4.18 Interface Curve

#### 6.4.18.1 Semantics

The simplest of the orientable primitives is the Curve. This interface contains elements needed for treating curves as geometric objects. In general, it must be kept in mind that there is always a way to choose any point interior to the curve (not the start or end point) and to break the curve at that point to create two curves, whose union is the equivalent geometry of the original curve. This is important in editing procedures where topology or **geometric complexes** are being maintained.

This document does not recognize "degenerate" nor "zero length" curves, which as geometry are points. Since many existing implementations do recognize such structures for technical reasons usually having to do with system procedures, care must be taken in such implementations to interpret geometries with a curve type but only point data.

**REQ. 110** **Implementations that recognized degenerate geometries (that do not satisfy their own dimensionality claims) of any type shall have operations that test whether a "local geometry instance" is actually of another topological dimension or matches the topological dimension claimed.**

**REQ. 111** **Any procedure requiring a non-empty, non-degenerate geometry of a particular topological dimension shall be allowed to refuse "degenerate" geometries that do not meet their dimensionality requirements.**

**REQ. 112** **A curve shall always be bounded in the sense that it shall have a finite envelop, i.e. not include "points at infinity".**

Curve (Figure 16) is a descendent subtype of Primitive through Orientable. It is the basis for 1-dimensional geometry. A curve is a continuous image of an open interval and so could be written as a parameterized function such as:

$$\boldsymbol{c}(\boldsymbol{t}):(\boldsymbol{a},\boldsymbol{b})\rightarrow\mathbb{R}^{\boldsymbol{n}} \tag{24}$$

where

"$t$" is a real parameter;

$\mathbb{R}^n$ and is coordinate space of dimension n (as determined by the geometric coordinate system).

Curves are continuous, connected and have a measurable length in terms of the coordinate system. The orientation of the curve is determined by this parameterization, and is consistent with the tangent function, which approximates the derivative function of the parameterization and shall always point in the "forward" direction.

The parameterization of the reversal of the curve $c(t):(a,b)\rightarrow\mathbb{R}^n$ would have a function of the form:

$$s(t)=c(a+b-t):(a,b)\rightarrow\mathbb{R}^{\boldsymbol{n}} \tag{25}$$

Any other parameterization that results in the same image curve, traced in the same direction, such as any linear shifts and positive scales such as

$$e(t)=c\left(a+t(b-a)\right):(0,1)\rightarrow\mathbb{R}^{n} \tag{26}$$

This is an equivalent representation of the same curve. For clarity, this document recognizes a construction parameter (usually written as "$t$") and an arc length parameter (usually written as "$s$"). The two symbologies $c(t)$ and $c(s)$ represent the same curve, one parameterized by the constructive parameter (from a "knot space") and the other by the arc length of the curve (measured from the start point to the location.

The derivative of a curve *c*, written as $\dot{c}$, is a tangent vector in the tangent space of the Reference Surface at the corresponding point on the curve. If the parameter is arc length, then the tangent is a unit tangent (of length 1):

$$\dot{c}(s) \in T(c(s)); \quad |\dot{c}(s)| = 1 \tag{27}$$

If the parameter is something other than arc length, then the direction of the tangent is the same but its length is changed by the "speed" of the other parameter.

$$\dot{c}(t) \in T(c(t)); \quad \dot{c}(t) = \|\dot{c}(t)\| \dot{c}(s) \; where \; c(t) = c(s) \tag{28}$$

If the coordinate space is $\mathbb{E}^3$, Euclidean 3-space (e.g. geocentric), then the curve is written in terms of the coordinates of the embedding $\mathbb{E}^3$ space, then if c is written using the knot space:

$$c(t) = (x(t), y(t), z(t)) \tag{29}$$

Using arc length, the variable “s” is used:

$$c(s) = (x(s), y(s), z(s)) \tag{30}$$

and:

$$\dot{c}(s) = \dot{c}(t) / \|\dot{c}(t)\| \; where \; c(t) = c(s) \quad \|\dot{c}(t)\| = \sqrt{(\dot{x}(t))^2 + (\dot{y}(t))^2 + (\dot{z}(t))^2} \tag{31}$$

The constructive tangent vector in the same $\mathbb{E}^3$ space is

$$\dot{c}(t) = (\dot{x}(t), \dot{y}(t), \dot{z}(t)) \tag{32}$$

The unit tangent is

$$\dot{c}(s) = (\dot{x}(s), \dot{y}(s), \dot{z}(s)) \tag{33}$$

In the case where the curve “c” uses any non-plane Geometric reference system, the second derivative $\ddot{c}(s)$ is perpendicular to the tangent. It can be written as a sum of a vector $\vec{k}_n$ normal to the surface and thus parallel to the surface normal $\vec{N}$, and a vector $\vec{k}_g$ tangent to the surface but normal to the curve's tangent "$\ddot{c}$", or using a unit vector $\vec{K}_n$ normal to the surface and a unit vector as $\vec{K}_g$ perpendicular to the tangent:

$$\ddot{c}(s) = \vec{k}_n + \vec{k}_g = \kappa_n \vec{K}_n + \kappa_g \vec{K}_g \tag{34}$$

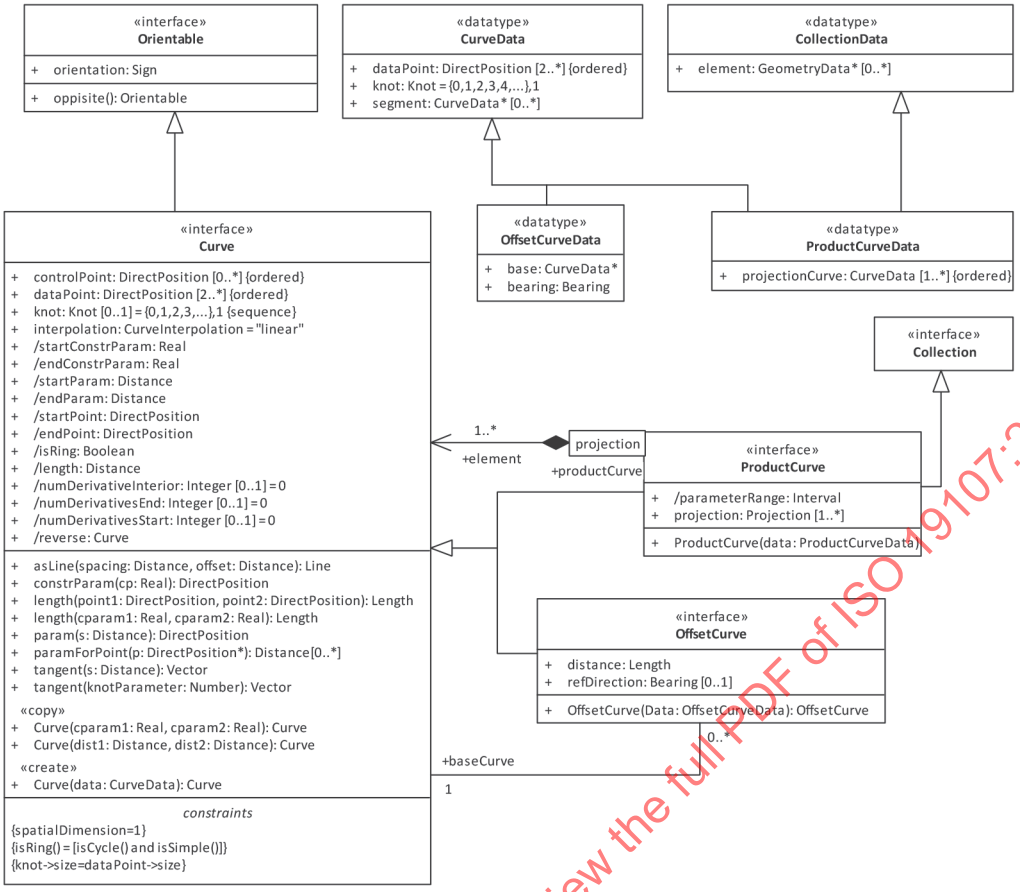



**Figure 16 — Curve, ProductCurve and OffsetCurve**

In these circumstances, the normal curvature vector $\vec{k}_n = \kappa_n \vec{K}_{\mathrm{n}}$ is dependent on the curvature of the surface, and the geodesic curvature vector $\vec{k}_g = \kappa_g \vec{K}_{\mathrm{g}}$ is from the curve itself. If the geodesic curvature is zero ($\kappa_{\mathrm{g}} = 0$) then the underlying curve is a geodesic.

A curve may be composed of one or more curve segments. Each curve segment within a curve may be defined using a different interpolation method. The curve segments are connected to one another, with the end of each segment except possibly the last being the start point of the next segment in the segment list. If the curve is a cycle, then the endpoint of the last is the start point of the first.

### 6.4.18.2 Attribute controlPoint

The controlPoint attribute contains a sequence of coordinate positions to be used in the construction of the curve geometry. The mechanism will depend on the curve type. The control points will have the same initial reference system as the geometry, but may have additional parameters columns for use in the calculation of the data points (see below).

```
Curve::controlPoint:DirectPosition[0..*]
```

EXAMPLE In a Rational spline, the control points are weighted by a homogeneous coordinate. In circles or circular strings, the additional parameter may indicate the bearing and distance from the control points (centre) to the data points.

#### 6.4.18.3 Attribute dataPoint

The array dataPoint contains a sequence of points on the curve. The first coordinate, dataPoint [0], will be equal to startPoint, and the last, dataPoint [dataPoint.Length -1], will be equal to the endPoint. The data points of a curve are in the coordinate space of the geometry.

```
dataPoint:DirectPosition[2..*]
```

#### 6.4.18.4 Attribute knot

Knots are constructive parameterization values that correspond to the dataPoint entries, so that, assuming an array k is the list of knot values, and P the list of dataPoint, and c the curve as a function of the construction parameter, then:

$$c(k_n) = P_n \tag{35}$$

The name "knot" is from spline theory; it means a point in the parameter space of $c(t)$ that is controlled by positions, in the controlPoint and dataPoint arrays. The term knot-space then means the domain from which knots are taken; i.e. the domain of the constructive parameter.

If the segment role is used to create a composite curve, then the knot arrays shall be consistent with one another:

If we define `"curve.knot.last"` as the last knot in the curve's knot array, and the `"segment.knot.last"`, `"segment.param.first"` and `"segment.param.last"` as the first or last knot or parameter used by a segments and all of its subsegments, then the continuity of a curve requires the following constraints:

```
{curve.startParam ≤ curve.endParam};

{curve.endParam = segment(1).param.first };

{∀i, segment(i).param.last = segment(i+1).param.first};

{∀i, segment(i).param.last = segment(i+1).param.first};

{∀i, segment(i).param.first ≤ segment(i).param.last};

{∀i, segment(i).knot.last = segment(i+1).knot.first};
```

#### 6.4.18.5 Attribute interpolation

The attribute "interpolation" specifies the curve interpolation mechanism used for the internal dataPoints of this object. Other segments, in the association role "segment" may choose other interpolation mechanisms. This mechanism uses the dataPoints, control points and control parameters to determine the position of this Curve. The default value is "linear" in the coordinate system of the object.

```
interpolation:CurveInterpolation="linear"
```

The attribute interpolation is a description of the mechanism using the controlPoint and dataPoint arrays to calculate the segment. The default value "linear" implies that the coordinate representation of each segment is a combination of two sequential values in the controlPoint array:

$$\vec{P}(t) = t\vec{P}_n + (1-t)\vec{P}_{n-1}; t \in [0,1] \tag{36}$$

#### 6.4.18.6 Attribute startPoint

The startPoint is the coordinate location of the beginning of the curve.

```
startPoint:DirectPosition
```

#### 6.4.18.7 Attribute endPoint

The endPoint is the coordinate location of the terminal end of the curve.

```
endPoint:DirectPosition
```

If the Curve uses segments, then the trancing of the curves internal data is first, followed recursively by each segment in the order of the segment relation. For this reason the following constraints are observed:

#### 6.4.18.8 Attribute isRing

The Boolean valued attribute “isRing” indicates whether the curve is both closed and simple. In this case, it would be a valid boundary component of a surface.

```
isRing:Boolean
```

#### 6.4.18.9 Attribute length

The attribute length is the length of the curve. The parameterization of the curve by arc length is thus on the interval [0, *length*].

```
length:Length
```

#### 6.4.18.10 Attribute numDerivativesInterior

The attribute numDerivativesInterior is the number of continuous derivative guaranteed for values of the construction parameter between the first and the last knot.

```
numDerivativesInterior:Integer [0..1]
```

#### 6.4.18.11 Attribute numDerivativesStart

The attribute numDerivativesStart is the number of continuous derivative guaranteed for values of the construction parameter just at the startPoint.

```
numDerivativesStart:Integer [0..1]
```

#### 6.4.18.12 Attribute numDerivativesEnd

The attribute numDerivativesEnd is the number of continuous derivative guaranteed for values of the construction parameter just at the endPoint.

```
numDerivativesEnd:Integer [0..1]
```

The attributes "numDerivativesStart" and "numDerivativesEnd" specify the type of continuity between this curve segment and its immediate neighbors, the first value for its predecessor, and the second for its successor. If this is the first or last curve segment in the curve, one of these values, as appropriate, is ignored. The attribute "numDerivativesInterior" specifies the type of continuity that is guaranteed interior to the curve. The default value of "0" means simple continuity, which is a mandatory minimum level of continuity. This level is referred to as "$C^0$" in mathematical texts. A value of 1 means that the function and its first derivative are continuous at the appropriate endpoint: "$C^1$" continuity. A value of "n" for any integer means the function and its first n derivatives are continuous: "$C^n$" continuity.

```
numDerivativesAtStart:Integer[0..1]=0;

numDerivativesInterior:Integer[0..1]=0;

numDerivativesAtEnd:Integer[0..1]=0;
```

NOTE Use of these values is only appropriate when the basic curve definition is an underdetermined system. For example, line strings and segments cannot support continuity above $C^0$, since there is no spare control parameter to adjust the incoming angle at the end of the segment. Spline functions on the other hand often have extra degrees of freedom on end segments that allow them to adjust the values of the derivatives to support $C^1$ or higher continuity. Functions to smooth curve segment transitions are important for merging segments while maintaining a level of continuity.

#### 6.4.18.13 Attribute startParam, endParam

The attributes startParam and endParam indicate the arc length parameters for the startPoint and endPoint respectively:

```
startParam:Length

endParam:Length

{param(startParam)=startPoint};

{param(endParam)=endPoint};

{length(startPoint,endPoint)=endParam-startParam=length}
```

The start and end parameter of a top level Curve (one which is not in the segment role of another curve) will normally be 0 and the arc length of the curve respectively. For segments within a Curve, the start and end parameters of the segment will normally be equal to those of the Curve where this segment begins and ends respectively in the segment role, so that the startParam of any segment (except the first) shall be equal to the endParam of the previous segment. If a Curve is used for other purposes, the two parameters must differ by the arc length of the Curve.

$$c(s):[startParam, endParam] \rightarrow \mathbb{R}^n \tag{37}$$

#### 6.4.18.14 Attribute reverse

The reverse of a Curve simply reverses the orientation of the parameterizations of the curve. In most cases, this involves a reversal of the parameters spaces in the curve segments, and a reversal of the order of the segments within a curve segmentation association inherited from primitive (0).

```
reverse:Curve
```

#### 6.4.18.15 Attribute startConstrParam, endConstrParam

The attributes startConstrParam and endConstrParam indicate the knot-space parameters for the startPoint and endPoint respectively. The "startConstrParam" and "endConstrParam" indicate the parameters used in the constructive parameterization for the startPoint and endPoint respectively:

```
startConstrParam:Real

endConstrParam:Real

constrParam(startConstrParam)=startPoint;

constrParam(endConstrParam)=endPoint;
```

There is no assumption that the startConstrParam is less than the endConstrParam, but the parameterization must be strictly monotonic (strictly increasing, or strictly decreasing).

$$c(t):[startConstrParam, endConstrParam] \rightarrow \mathbb{R}^n \tag{38}$$

NOTE Constructive parameters are often chosen for convenience of calculation, and seldom have any simple relation to arc distances, which are defined as the default parameterization. Normally, geometric constructions will use constructive parameters, as the programmer deems reasonable, and calculate arc length parameters when queried.

#### 6.4.18.16 Attribute boundary

The inherited attribute "boundary" on Curve returns the geometry of its topological boundary. For simple curves, if startPoint is not equal to endPoint, the boundary is a two-point array. If startPoint is equal to endPoint, the boundary is an empty array.

```
boundary:Point[0..2]
```

If the curve is not simple, it may have only one boundary point depending on the choice of boundary (see 10.8.3). For example, a rho curve (shaped like the Greek letter “ρ”) may be a single point or two points depending on the choice of boundary paradigm.

NOTE The above point array will usually be two distinct positions, but both Curves can be cycles in themselves. The most likely scenario is that all of the points used will be transients (constructed to support the return value), except for the startPoint and endPoint of the aggregated Curve. These two positions, in the case where the Curve is involved in a Geometry Complex, will be represented as Points in the same Geometry Complex. In earlier versions of this document, this operation was defined separately on the subclasses of Curve.

#### 6.4.18.17 Operation asLine

The function "asLine" determines an array of Points lying on the original curve that are used to constructs a line string. If "spacing" is given and not zero, then the distance between the points in this sequence shall be not more than "spacing". If "offset" is given and not zero, the distance between the positions on the Line and the original curve shall not be greater than the "offset". If both parameters are set, then both criteria shall be met. If the original control points of the Curve lie on the curve, then they shall be included in the points in the Line's controlPoint array. If both parameters are zero, then the implementation may use a general "accuracy" limit universal for the data.

```
asLine(spacing:Distance=0,offset:Distance=0):Line
```

The "spacing" parameter will control an upper limit of the distance on the original curve of any two data points on the line string. A "0" means no limit is given. The "offset" parameter will control the upper limit of the distance of any point on the original curve between two data points that are preserved in the line string, to some point on the original curve between these same two points. Again, a zero limit means no limit.

**REQ. 113** **When invoking the operation "asLine" with both zero spacing and offset limits, the resulting line shall include at least all of the DirectPositions in the dataPoint array of the original curve.**

**REQ. 114** **All points on the resulting line shall be on the original curve within the accuracy of the numerical representation for the direct positions.**

NOTE This function is useful in creating linear approximations of the curve for simple actions such as display. It is often referred to as a "stroked curve". For this purpose, the "offset" version is useful in maintaining a minimal representation of the curve appropriate for the display device being targeted (offset = pixel radius). This function is also useful in preparing to transform a curve from one coordinate reference system to another by transforming its control points. In this case, the "spacing" version may be more appropriate.



#### 6.4.18.18 Operation constrParam

The operation constrParam maps the "convenience" parameterization in the knot space to positions on the curve. Knot values will produce entries in the dataPoint array. The operation may be an alternate representation of the curve as the continuous image of a real number interval without the restriction that the parameter represents the arc length of the curve, nor restrictions between a Curve and its component Curves. The most common use of this operation is to expose the constructive equations of the underlying curve, especially useful when that curve is used to construct a parametric surface. The default values of the knot space are an array of integer valued, counting from 0, by 1, up to the number of "knots".

```
constrParam(cp:Real):DirectPosition
```

#### 6.4.18.19 Operations length

The operation length will return the length of the curve between two points on the curve. If the curve is not simple and passes through the positions more than once, the length returned will be the smallest lengths for the curve between the two points. Another variant of length uses constructive parameters (from the knot space) to determine the segment of the curve for which a length is to be calculated.

```
length(point1:DirectPosition=startPoint,
point2:DirectPosition=endPoint):Length

length(cparam1:Real=startConstrParam,
cparam2:Real=endConstrParam):Length
```

The length of a piece of curvilinear geometry shall be a numeric measure of its length in a coordinate reference system. Since length is an accumulation of distance, its return value shall be in a unit of measure appropriate for measuring distances. The operation "length" shall return the distance between the two points along the curve. The default values of the two parameters shall be the start point and the end, respectively. If either of the points is not on the curve, then it shall be projected to the nearest DirectPosition on the curve before the distance is calculated. If the curve is not simple and passes through either of the two points more than once, the distance shall be the minimal distance between the two points on this curve.

The second form of the operation length shall work directly from the constructive parameters, allowing the direct conversion between the variables used in parameterization and constrParam.

Distances between DirectPositions determined by the default parameterization are simply the difference of the parameter. The length function also allows for the conversion of the constructive parameter to the arc length parameter.

```
If p=(startParam + length(startConstrParam, p2))
then param(p)=constrParam(p2)
```

#### 6.4.18.20 Operation param

**REQ. 115** **The operation "param" shall be the parameterized-by-length representation of the curve as the continuous image of a real number interval. The operation returns the DirectPosition on the Curve at the distance passed. This parameterization shall be by arc length, i.e. distance along the Curve measured from the start point and added to the start parameter.**

```
param(s:Length):DirectPosition
```

#### 6.4.18.21 Operation paramForPoint

The operation paramForPoint returns a parameter for the "arc length" interpolation of the curve that most closely returns the original point. If the DirectPosition is not on the curve, the nearest point on the curve shall be used.

```
paramForPoint(p:DirectPosition):Distance[0..*]
```

If the passed point " p:DirectPosition " is on the curve, and the returned distance is "d:distance" then:

```
param(d)==p and paramForPoint(p) contains d
```

The DirectPosition closest is the actual value for the "p" used, that is, it shall be the point on the Curve closest to the coordinate passed in as "p". The return set will contain only one distance, unless the curve is not simple. If there is more than one DirectPosition on the Curve at the same minimal distance from the passed "p", the return value may be an arbitrary choice of one of the possible answers.

#### 6.4.18.22 Operation tangent

The overloaded operation tangent(number) takes a knotParameter and returns a tangent vector associated with the coordinate system at the point whose constructive parameter was given on the curve. If the coordinates in the direct positions are ($x_1$, $x_2$, $x_3$....) then the coordinates for the vector are ($dx_1$, $dx_2$, $dx_3$....) the differential operators for the coordinates. If $\vec{\tau}$ is this tangent, and the unit tangent is $\vec{\mu}$ then

$$\frac{dc}{dt}=\vec{\tau};\qquad \frac{dc}{ds}=\vec{\mu}$$
$$\vec{\mu}=\frac{\vec{\tau}}{\|\vec{\tau}\|}\Rightarrow\|\vec{\tau}\|=\frac{ds}{dt}\Rightarrow\frac{1}{\|\vec{\tau}\|}=\frac{dt}{ds} \tag{39}$$

where

$t$ is the knot parameter;

$s$ is the arc length parameter.

```
tangent(knotParameter:Number):Vector

tangent(s: Length): Vector
```

The operation tangent(Length) returns a tangent vector associated with the coordinate system at the point "s" arc length parameter of the curve. If the coordinates in the direct positions are ($x_0$, $x_1$, $x_2$, $x_3$....) then the coordinates for the vector are multiples of ($dx_0$, $dx_1$, $dx_2$, $dx_3$....) the differential operators for the coordinates. This vector approximates the derivative of the parameterization of the curve. The tangent with respect to arc length will be a unit vector (have length 1.0), which is consistent with the parameterization by arc length.

```
tangent(s:Distance):Vector

Copy Constructor Curve(cparam1:Real, cparam2:Real):Curve

Copy Constructor Curve(dist1:Distance,Distance:Distance):Curve
```

The overloaded copy constructor Curve, takes two points on an existing curve and excises the part of that curve from the first given position to the second positions. The first version uses the constructive parameter to select points. The second version uses distance "param". If the first parameter is less than

the second, then the new Curve will agree in orientation with the original one. If the first parameter is greater than the second, the orientation will be reversed.

```
Curve(cparam1:Real, cparam2:Real):Curve

Curve(dist1:Distance, Distance:Distance):Curve
```

### 6.4.19 DataType CurveData

The datatype CurveData, a subtype of OrientableData, will contain copies of the attributes needed for every Curve construction. The attributes are described in 6.4.18.5 "interpolation" which is usually derivable from the class name, 6.4.18.4 "knot", the lists of dataPoints, 6.4.18.3, as DirectPositions, and the list of segments for any subsegments as CurveData. Some curve interpolations will require controlPoints also (e.g. b-spline).

### 6.4.20 Interface OffsetCurve

#### 6.4.20.1 Semantics

An offset curve is a curve at a predictable distance and bearing from a base curve. They can be useful as a simple alternative to constructing curves that are offsets by definition. For example, if a highway's centreline is stored, then offset from that curve at a constant distance is the right hand and left hand side of the road.

The offset curve stores the offset and is can be constructed from the data or control points of the original curve. The offset is a bearing and a distance. If the bearing is absolute, the offset is the geographic equivalent of a parallel translation. If the offset is relative, then the base direction is the tangent of the original curve, and the offset is a motion referenced to the base curve's direction. For example, if the distance is 10 metres, and the bearing is 90°, then the offset curve is the right-hand side of the 10 metre buffer of the original centreline (the small loops generated as in the usual buffer algorithm are removed as they are in buffer generation, and each point on the offset curve is exactly the correct distance from the original curve).

#### 6.4.20.2 Attribute distance: Length

The attribute "distance" is the distance at which the offset curve is generated from the basis curve.

```
distance:Length
```

NOTE This assumes a constant distance from the base curve. Extensions to this document can address this issue by using a function of position to determine distance (see ISO 19148 and the discussion of linear reference systems, and ISO 19141 and the discussion on moving objects). Such approaches can require augmenting the coordinate system with a linear reference to make the offset function easier to define. There is a general concept that coverage functions that map positions to values can be used to replace attribute values using the range type of the function. Such extensions are consistent with this document, but require mechanisms of their own that are currently out of the scope of this document. In this case, the attribute distance could be replaced by a function of position along the base curve. Such an approach might look like the following, where the domain of the returned coverage function contains the base curve:

#### 6.4.20.3 Attribute refDirection

The attribute "refDirection" is used to define the vector direction of the offset curve from the basis curve. It can be omitted in cases where the spatial dimension is 2. The distance can be positive or negative. In the 2D case, distance defines left side (positive distance) or right side (negative distance) with respect to the tangent to the basis curve.

In 3D, the base curve shall be required to have a well defined tangent direction for every point. If the curve is not differentiable at some points, then the application may use a reasonable approximation to a

smoothly varying tangent vector. The offset curve at any point (parameter) on the basis curve "c" is in the direction:

$$\vec{s} = \vec{v} \times \vec{t} \text{ where } \vec{v} = c.refDirection \text{ and } \vec{t} = \dot{c} = c.tangent \tag{40}$$

For the offset direction to be well-defined, the refDirection $\vec{v}$ shall not at any point of the curve be in the same, or opposite, direction as $\vec{t}$; which usually means that the curve does not have a purely vertical slope at any point.

refDirection:Bearing[0..1]

NOTE The Bearing may not be constant. For example, if the reference direction is a variable tied to the underlying curve (e.g. left, right), then the Bearing will vary along the base curve position.

#### 6.4.20.4 Association role: baseCurve: Curve

The association role "baseCurve" is a reference to the curve from which this curve is defined as an offset. All non-positional attributes, such as coordinate system, for the offset curve must be the same as for the base curve.

baseCurve:Curve

### 6.4.21 Datatype OffsetCurveData

The datatype, OffsetCurveData a subtype of CurveData, will contain in addition to its inherited attributes, attributes described in Clause 0 "length", and 0 "bearing" and an attribute of type CurveData* refereeing the base curve, described in 0.

### 6.4.22 Interface ProductCurve

#### 6.4.22.1 Semantics

A product cure is a curve composed of other curves each sharing the same parameter space. For example, a product curve "$c$" built from 2 projection curves "$c_1$, $c_2$" would work like the following:

$$\left[c_1 : [a,b] \to \mathbb{R}^n ; c_2 : [a,b] \to \mathbb{R}^m\right] \Rightarrow \left[c = c_1 \otimes c_2 : [a,b] \to \mathbb{R}^{n+m}, c(t) = (c_1(t), c_2(t))\right] \tag{41}$$

The more general form would be:

$$\left[0 \le i \le n, c_i : [a,b] \to \mathbb{R}^{n_i}\right] \Rightarrow \left[c = \left(\underset{i}{\otimes} c_i\right) : [a,b] \to \mathbb{R}^{\sum_i n_i}\right] \tag{42}$$

In the extreme case, each dimension in the geometric coordinate space would be defined by a separate function. A unit circle as a product curve in $\mathbb{R}^2$ would be $c(t) = (sin(t), cos(t))$.

A product curve may have different interpolation mechanisms for various disjoint subsets of the coordinate offsets in the CRS. As such, it is both a collection of curves in various projections of the CRS, and a curve in the full CRS. The collected curves support various offsets in the CRS, passing through the projections of the datapoints and sharing the same knots. The ProductCurve and its projections share constructive parameters.

#### 6.4.22.2 Attribute parameterRange: Interval

ProductCurve inherits a knot array from Curve, the first and last knot is the interval of the curve.

**REQ. 116** **In a ProductCurve, all projected elements shall be Curves. These Curves shall be in the elements array inherited from Collection. Each of these Curves shall be a projection of the ProductCurve into a disjoint projection of the coordinate system of the ProductCurve. Each of these projected Curves shall share the knot array of the ProductCurve. Each coordinate offset of the Product Curves coordinate system shall be contained in one and only one of these elements.**

```
parameterRange: Interval
```

### 6.4.22.3 Attribute projection: Projection [1..*]

The attribute projection is the set of projections of the coordinate system of the ProductCurve that correspond, in order, to the coordinate systems of the element curves. This may be redundant to the coordinate systems in the element association where each element carries its own coordinate system "rsid". Thus, it is an application implementation option since the projections can be derived.

```
projection: Projection[1..*]
```

### 6.4.22.4 Aggregation role: element

For each set of coordinate offsets that share an interpolation, the ProductCurve has a projection curve into that subset. For example, any curve could be written as a ProductCurve whose projections are a set of 1D curves in each of the coordinate offsets, as in the formula:

$$\vec{c}(s)=\big(x(s),y(s),z(s)\big) \tag{43}$$

The role `elements:Curve[]` lists all of the projected curves whose composite is the Product Curve.

```
(element, projection):Curve[1..*]
```

NOTE In some cases, the interpolations of the coordinates can be cross dependent. So if there is a digital elevation model (DEM) $z = h(x, y)$ h( that associates a z-value for any (x, y) position, there would be a 3D ProductCurve following the elevation model defined by any 2D curve in (x, y):

$$\vec{c}(s)=\big(x(s),y(s),h\big(x(s),y(s)\big)\big) \tag{44}$$

### 6.4.22.5 Constructor: ProductCurve(data: ProductCurveData)

A ProductCurve is essentially a collection of curves, whose domain is the various projections of the coordinate space for the full curve.

```
ProductCurve: ProductCurve(data: ProductCurveData)
```

EXAMPLE In 3D a curve can be written as $c(t) = [x(t), y(t), z(t)]$ so the construction of C from 3 curves in "1-space" would be something like this:

$$C=\text{ProductCurve}\big(\text{CRS},\big((\text{CRS(x)},X),(\text{CRS}(y),Y),(\text{CRS}(z),Z)\big)\big) \tag{45}$$

The Collection datatype inherits a CRS reference from the GeometryType, which can then be used in its projection to each dimension to take the curves (in 1D) X, Y, and Z, synchronized by the common variable "t" into a "3D Curve".

### 6.4.23 ProductCurveData

The DataType ProductCurve Data, in addition to attributes inherited from CurveData will contain an array for the projected curve using CurveData datatypes. Unlike Curve, this array is not a set of references. The rsid attribute of the constructed product curve will be concatenation of all the rsid arrays in the projected curves.

### 6.4.24 CodeList: CurveInterpolation

CurveInterpolation is a list of codes that may be used to identify the interpolation mechanisms specified by an application schema. As a codelist, there is no intention of limiting the potential values of CurveInterpolation. Subtypes of Curve can be spawned directly through subclassing, or indirectly by specifying an interpolation method and a default constructor control parameters datatype to support it. All interpolations used in this document have well-known algorithms for splitting a curve segment at any point on the curve and maintaining the underlying geometry object (as a set of DirectPositions). This becomes very important in maintaining topology, and should be maintained (at least to a controllable degree of accuracy) by any extension to this document that adds interpolation types.

**REQ. 117** **The set of curve interpolations shall be implemented by a codelist. The local codelist CurveInterpolation shall only contain interpolation methods supported by the local application.**

**Table 2 — Codelist CurveInterpolation by interface and default constructor data**

| Interpolation | Curve Interface | Constructor Datatype |
|---|---|---|
| compositecurve | Curve | CurveData |
| productcurve | ProductCurve | ProductCurveData |
| linear | Line | LineData |
| geodesic | Geodesic | GeodesicData |
| rhumb | Rhumb | RhumbData |
| circular | Arc, Circle | ArcData, CircleData |
| spiral, clothoid | Spiral, Clothoid | SpiralData |
| elliptical | EllipticArc | EllipticArcData |
| conic | Conic | ConicData |
| polynomialspline | PolynomialSpline, CubicSpine | PolynomialCurveData |
| bezierspline | Bezier (BSpline) | BSplineData |
| bspline | BSpline | BSplineData |
| nurbs | Bspline in homogeneous coordinate systems | BSplineData |
| product | ProductCurve | ProductCurveData |
| composite | Curve | CurveData |

**Table 3 — Code values and definitions for CurveInterpolation**

| Code Value | Definition |
|---|---|
| linear | The linear interpolation mechanism returns DirectPositions on a straight line between each consecutive pair of dataPoints. |
| geodesic | The geodesic interpolation mechanism returns DirectPositions on a geodesic curve between each consecutive pair of dataPoints. A geodesic curve is a curve of shortest length. The geodesic shall be determined in the coordinate reference system of the Curve in which the Curve is used. |

**Table 3** *(continued)*

| Code Value | Definition |
|---|---|
| circular | The circular interpolation mechanism returns DirectPositions on a circular arc passing through a set of dataPoints. The sequence of control points shall each be a circle centre such that that circle passes through two consecutive data points, and touch with the previous (if not the first arc) and the next (if not the last arc). The radius of the circle is determined by the geodesic distance from the control point to the first of its dataPoints. The second data point should be less than 180° of an arc from the first. This disambiguates the direction of the arc; longer arcs in the same may use other dataPoints with a repetition of the centre in the controlPoints. The second continuous arcs may be at any offset in that same direction of rotation (reversal of direction is not allowed). Circles are generated using the geodesic distance. A circle in the tangent plane at the centre point on a Reference Surface of the given radius can mapped onto a circle on this surface by the exponential map. |
| spiral | One of the classical spirals is generated at each control point in such a manner so that they form a continuous curve. The spiral type, beginning and ending points and parameters is specified for each control point; the spiral is generated in the tangent space at the control point and mapped onto the surface by the exponential map just as with the circle |
| clothoid | Uses a Cornu's spiral or clothoid interpolation, a specialization of spiral. |
| elliptical | Ellipses are generated in a manner similar to circles in the tangent space at the ellipse's centre and projected on the Reference Surface using the exponential map. |
| conic | Same as elliptical arc but using five consecutive datapoints in the tangent space to determine a conic section. |
| polynomialSpline | The dataPoints are ordered as in a line-string, but they are spanned by a polynomial function. Normally, the degree of continuity is determined by the degree of the polynomials chosen. |
| bezierSpline | The data are ordered as in a line string, but they are spanned by a polynomial or spline function defined using the Bézier basis. Normally, the degree of continuity is determined by the degree of the polynomials chosen |
| bSpline<br>NURBS | The controlPoints are ordered as in a line string, but they are spanned by a polynomial or rational (quotient of polynomials) spline function defined using the B-spline basis functions (which are piecewise polynomials). The use of a rational function is determined by the Boolean flag "isRational". If isRational is TRUE then all the DirectPositions associated with the control points are in homogeneous form. Normally, the degree of continuity is determined by the degree of the polynomials chosen. |
| product | The interpolation is done differently on various projections, i.e. offsets in the coordinates. |

Valid names for the set of interpolations may include, but are not limited, to the values in Table 3. The value of the interpolation mechanism is restricted by interface type to the values in Table 2.

### 6.4.25 Interface Surface

#### 6.4.25.1 Semantics

Surfaces are the elements of geometry of two topological dimensions.

**REQ. 118** **A surface shall be connected (contiguous) in such a manner that for any two points on the surface, but not on its boundary, it is possible to create a curve from one to the other in such a manner that the curve is also completely contained in the interior of the surface.**

**REQ. 119** **A surface shall have as its boundary a set of simple closed curves (called rings). The surface is to the left of all of its rings. These rings shall not self-intersect or be self-tangent.**

EXAMPLE A line from (0, −180) to (0, 180) is defined by linear interpolation so would have an equation representation of $c$:[0,1] →{($\varphi$,$\lambda$)} by the vector equation c(x) = (1 − x)[0, −180]+ x[0, 180], which begins as [0, −180], with $\varphi$ = 0 and $\lambda$ traversing −180 to +180. By REQ. 10, the start point is the same as the endpoint, but the linear interpolation of a line only looks at the numbers, so this line is the equator starting at the "international date line" on the equator and moves east until it gets back to where it started. This line is a simple closed curve and fits the definition of a ring. The northern hemisphere is to the left. This is a valid linear curve (a Line, see 7.1.2; it is also a Rhumb line, but that is accidental as most lat-long lines are not loxodromes) boundary for the northern hemisphere.

There may for rings in the boundary of the same surface, be tangent to one another at a single point, left side to left side. As a set and within the limits of calculation error on the machine in question, the boundary of a surface is uniquely representable by rings.

**REQ. 120 A surface in shall always be bounded in the sense that it has a finite envelop, i.e. it does not include "points at infinity".**

Surfaces in 2D will always have a planar interpolation because of the restriction of the dimension of the coordinate space. In 2D because of the restriction above, a surface will have a unique exterior ring, the one with the largest envelope, and some number of interior rings. Each such ring in 2D satisfies the Jordan Curve Theorem and thus divides the space into exactly two regions, one bounded and one unbounded. For rings that are counter clockwise, the bounded area is to the left of the ring as a curve. For rings that are clockwise, the bounded area is to its right. Each ring is said to define an area, the one to its left. A surface in 2D is the set intersection of the areas defined by its rings.

In all cases, the rings in a boundary of a surface shall satisfy the following criteria:

**Table 4 — Surface restrictions**

| |
|---|
| They will all be simple, with no self-intersections and no self-tangencies. |
| They will all be closed, with the start point equal to the endpoint. |
| They may be locally tangent to one another, but only once for each pair, only at a point, and only left side to left side. |
| The interior of a surface must be path connected (see REQ. 118) |

**REQ. 121 For an instance of a Surface, its boundary rings shall satisfy semantic restrictions in Table 4 — Surface restrictions.**

Surface (Figure 17) a subclass of Geometry Primitive and is the basis for 2-dimensional geometry. Unorientable surfaces such as the Möbius band are not allowed as surfaces, but may be constructed as a type of Geometry Complex. The orientation of a surface chooses an "up" direction through the choice of the upward normal, which, if the surface is not a cycle, is the side of the surface from which the exterior boundary appears counterclockwise. Reversal of the surface orientation reverses the curve orientation of each boundary component, and interchanges the conceptual "up" and "down" direction of the surface. If the surface is the boundary of a solid, the "up" direction is "outward". For closed surfaces that have no boundary, the up direction is that of the surface segments which must be consistent with one another. The Surfaces instance's included Surface describes the interior and interpolative structure of a Surface.

Other than the restriction on orientability, no other "validity" condition is required for Surface.

### 6.4.25.2 Attribute boundary: Curve [1..*]

The attribute boundary will contain Curves that have the Surface on their left hand side. The first Curve may be considered at the exterior boundary, but with the exception of the plane, the concept of an unbounded exterior does not apply.

```
Surface::boundary:Curve[1..*]
```

In many cases, the boundary curves are derived from the surface, but they may be specified independently of the surface and then used in its construction. For example, the boundary curves in a 2D coordinate system are sufficient to determine completely the "polygon" surface they bound.

#### 6.4.25.3 Attribute interpolation

The attribute "interpolation" determines the surface interpolation mechanism used for this Surface. This mechanism uses the control points and control parameters defined in the various subclasses to determine the position of this Surface.

```
Interpolation:SurfaceInterpolation[0..*]="polygon"
```

#### 6.4.25.4 Attribute numDerivativesBoundary: Integer [0..1]

The optional attribute "numDerivativesBoundary" specifies the type of continuity between this surface and its immediate neighbors with which it shares a boundary curve. The default value of "0" means simple continuity, which is a mandatory minimum level of continuity. This level is referred to as "$C^0$" in mathematical texts. A value of one means that the functions are continuous and differentiable at the appropriate ends "$C^1$" continuity. A value of "*n*" for any integer means n-times differentiable: "$C^n$" continuity.

```
numDerivativesOnBoundary:Integer[0..1]=0
```

As with curves, any surface and therefore any surface segments (or the older term "patch") shall be splitable along any curve on the surface that cuts the surface into at least two distinct pieces. This is not always easily done, and the orientable proxy may be required. For this reason, any surface segment should have a parameterization from some subset of 2D Euclidean space. This will be discussed at length in the clauses on the various subtypes of surface.

#### 6.4.25.5 Attribute numDerivativeInterior: Integer [0.., 1]

The optional attribute numDerivativeInterior is the minimal level of continuity within the surface's interior.

```
numDerivativeInterior:Integer[0..1]=0
```

#### 6.4.25.6 Attribute perimeter: Length

The attribute perimeter will be the sum of the lengths of all boundary curves.

```
perimeter:Length
```

#### 6.4.25.7 Attribute area: Area

The attribute area is the total area of the surface.

```
area:Area
```

#### 6.4.25.8 Attribute dataPoint: DirectPosition [0..*]

The potentially attribute dataPoint is a collection of points on the surface.

```
dataPoint:DirectPosition[1..*]
```

#### 6.4.25.9 Attribute controlPoint: DirectPosition [0..*]

The attribute controlPoint is a collection of points that are used in the construction of the surface, based on the interpolation type.

```
controlPoint:DirectPosition[1..*]
```

#### 6.4.25.10 Attribute knot: Knot[0..1]

If the constructive parameter space cannot be the default, then, a pair of lists of knot values is included, see 6.4.17.

#### 6.4.25.11 Operation: upNormal (point: DirectPosition): Vector

The operation upNormal will return a vector perpendicular to the surface at the point. The Normals on the surface will be consistent (the surface will be two sided). In the case where the coordinate system is 2D, the Normal will be in the 3D embedding Cartesian coordinate space of the Geometric reference system. In a 3D coordinate system, the upNormal is a vector at the point perpendicular to the tangent plane of the surface at the point.

```
upNormal(point:DirectPosition):Vector
```

This requires that the surface is orientable, and therefore not some variant of a Möbius Band or Kline Bottle.

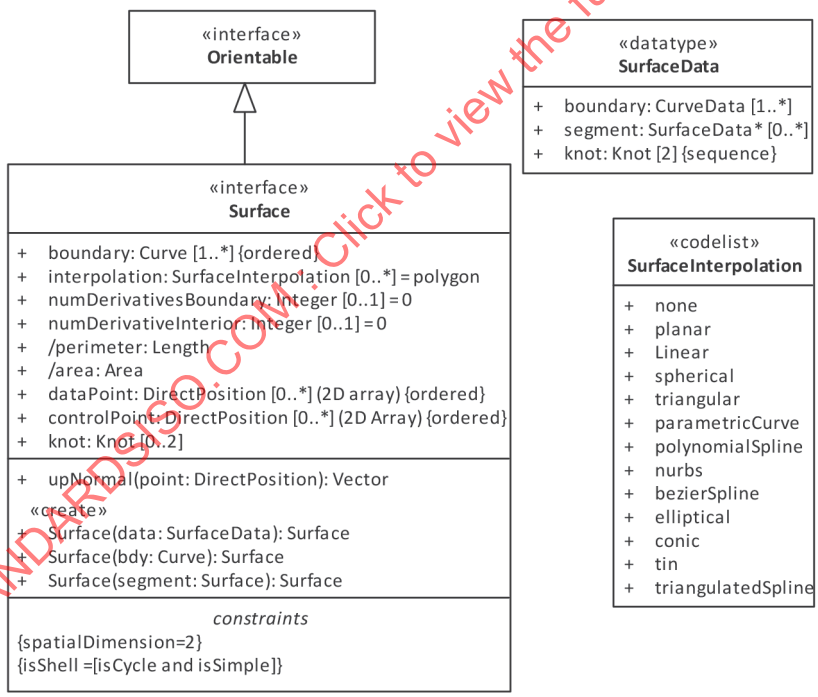



**Figure 17 — Surface**

#### 6.4.25.12 Operation: Surface (constructors)

The first version of the constructor for Surface takes a list of surfaces with the appropriate side-to-side relationships and creates a Surface.

```
Surface(segment:Surface[1..*]):Surface
```

The second version, which is guaranteed to work always in 2D coordinate spaces, constructs a Surface by indicating its boundary as a collection of Curves organized into a collection of simple, closed curves. In 3D coordinate spaces, this second version of the constructor shall require all of the defining boundary Ring instances to be coplanar (lie in a single plane) which will define the surface interior interpolation method as planar.

```
Surface(boundary:Curve[1..*]):Surface
```

### 6.4.26 Datatype SurfaceData

The datatype SurfaceData, a subtype of OrientableData, will contain copies of the attributes needed for every Surface construction. The attributes are describe in Clause 0 "interpolation", and 0 "boundary", 0 "dataPoints", 0 "controlPoints" (if needed) and an attribute "segment" of type "SurfaceData*[0,*]" referencing SurfaceData instances representing the associated segments defined in Clause 0 "segment".

### 6.4.27 CodeList: SurfaceInterpolation

SurfaceInterpolation (Figure 17) is a list of codes that may be used to identify the interpolation mechanisms specified by an application schema. Valid values for "interpolation" include, but are not limited, to the following:

If more than one interpolation description fits the method used, then the most restrictive one will be used.

**Table 5 — SurfaceInterpolation values**

| VALUE | DEFINITION |
|---|---|
| none | The interior of the surface is not specified. The assumption is that the surface follows the Reference Surface defined by the coordinate reference system. |
| planar | The interpolation method returns points on a single plane. The boundary in this case shall be contained within that plane. |
| linear<br>bilinear | Control points (some of which may be outside of the surface boundary) are organized into 2D arrays, and each square in the array uses the unit square in $\mathbb{E}^2$ . $\{(x, y) \mid 0 \le x \le 1, 0 \le y \le 1\}$ then interpolating using bilinear interpolation. |
| spherical<br>elliptical<br>conic | The surface is a section of a spherical, elliptical or conic surface. |
| triangular<br>triangle | The control points are organized into adjoining triangles, which form small planar segments. |
| parametricCurve | The control points are organized into a 2-dimensional grid and each cell within the grid is spanned by a surface that shall be defined by a family of curves. |
| polynomialSpline | The control points are organized into an irregular 2-dimensional grid and each cell within this grid is spanned by a piecewise polynomial spline function. In the case of Bézier or b-splines, the more specific term is used. |
| bspline | The control points are organized into an irregular 2-dimensional grid and each cell within this grid is spanned by a basis spline function. |
| nurbs | A b-spline in homogeneous coordinates, the associated Boolean flag "isRational" will be set to true. |
| bezier | The control points are organized into irregular 2-dimensional grid, each cell within this grid is spanned by a Bézier spline function. |

### 6.4.28 Interface Solid

#### 6.4.28.1 Semantics

The Solid primitive contains interfaces needed for treating Solid as geometric objects. Since a solid is by definition, a 3D topological object, it can only exist in a coordinate space with three spatial dimensions.

In a 3D world, it is sufficient to define surfaces by their boundary (B-REP or boundary representation). In cases of higher dimensions, such as may be found in coverage functions, some types of solid modelling with deal with non-homogeneous interior structures or in cases where parameterizations are used to map "template geometries" into real space, a more complex but more capable approach may be useful. In these cases, the structure of the interior of the solid may be represented by functions based on the same sort of mathematics used to embed surfaces in 3D space.

In all cases, a solid has a boundary consisting of surfaces in the same coordinate system as the solid. These surfaces can be organized into shells that are the 3D equivalent to the rings for a surface. A solid shall be connected (contiguous) in such a manner that for any two points on the solid, but not on its boundary, it is possible to create a curve from one to the other in such a manner that the curve is also completely contained in the interior of the solid. A solid shall have as its boundary a set of simple closed surfaces (shells). A solid is always below it shells (as defined by the upward normal of the surface). These shells shall not self intersect nor be self tangent. They may, for shells in the boundary of the same solid, be tangent to one another at a single point or along a non-closed, simple curve, bottom side to bottom side (that means that the upward normals of the two at points of contact are in opposite directions). As a set and within the limits of calculation error on the machine in question, the boundary of a solid is uniquely representable by shells (as a collection of surfaces that are both cycles (closed) and simple).

**REQ. 122 A solid shall always be bounded in the sense that it has a finite envelop, i.e. not include "points at infinity".**

Solids in 3D will always have a simple interpolation because of the restriction of the dimension of the coordinate space. In 3D because of the restriction above, a solid will have a unique exterior shell, the one with the largest envelope, and some number of interior shells. Each such shell in 3D satisfies the Jordan-Schönflies (Jordan Separation) Theorem and thus divides the space into exactly two regions, one bounded and one unbounded. Each shell is said to define a volume, the one in the direction opposite from the upward normal.

Solid (Figure 18), a subclass of Geometry Primitive, is the basis for 3-dimensional geometry. The spatial extent of a solid is defined by the boundary surfaces.

#### 6.4.28.2 Attribute boundary: Surface [1..*]

In instantiations of this interface, which will always be instantiations of Geometry, the attribute "boundary" specializes the boundary defined at Geometry and at Geometry Primitive with the appropriate return type.

**REQ. 123 The boundary of a Solid shall be a set of Surfaces that limit the extent of this Solid**

These surfaces shall be organized into one set of surfaces for each component of the solid. Each of these shells will be a cycle (closed composite surface without boundary).

```
Solid::boundary:Surface[1..*]
```

NOTE The exterior shell of any single component solid is defined only because the embedding coordinate space is always a 3D Euclidean one; i.e. the 3D space in which the GeometricReferenceSurface for its coordinates is defined. In general, a solid in a bounded 3-dimensional manifold has no distinguished exterior boundary.

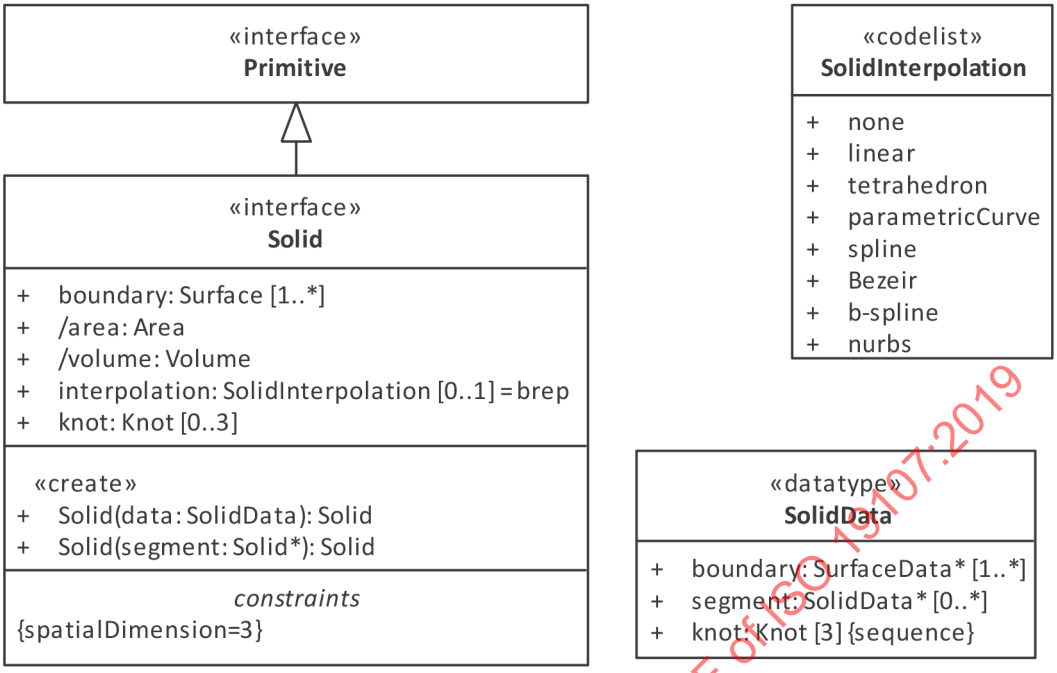



**Figure 18 — Solid**

In cases where "exterior" boundary is not well defined, all the shells of the solid boundary shall be listed as "interior". This can only happen in a minimum of 4 spatial dimensions and is beyond the scope of this document.

The shells that bound a solid shall be oriented outward – that is, the "top" of each Shell as defined by its "upward normal" orientation shall face away from the interior of the solid.

#### 6.4.28.3 Attribute area: Area

The attribute "area" shall specify the sum of the surface areas of all of the boundary components of a solid.

```
Solid::area:Area
```

The array class Surface has a "column operation" called "area" that accumulates the area of the components of the set. Using this, it can be said that for a Solid:

```
Solid:area=self.boundary.area
```

#### 6.4.28.4 Attribute volume: Volume

The attribute "volume" shall specify the volume of this solid. This is the volume interior to the exterior boundary shell exterior to any interior boundary shell.

```
Solid::volume:Volume
```

#### 6.4.28.5 Attribute dataPoint: DirectPosition [0..*]

The optional dataPoint array can carry point sample information about the interior and boundary of the solid. In general, the dataPoints can be treated as a "point cloud" of information, but the various subtypes of solid can add requirements on this array. The directPosition values of this array can carry as many "parameter" columns as required by the reference system associated with the object as geometry in 6.2.4. The interpolation mechanism can be found in the interpolation attribute in 6.4.29.

```
Solid::dataPoint:DirectPosition[0..*]
```

#### 6.4.28.6 Attribute controlPoint: DirectPosition [0..*]

The optional controlPoint array can often be used similarly to the dataPoint array, except controlPoint positions do not need to be contained in the solid. The interpolation values from control points are only valid if they are contained in the boundary of the solid.

```
Solid::controlPoint:DirectPosition[0..*]
```

#### 6.4.28.7 Attribute interpolation: SolidInterpolation [0..1]

The optional attribute interpolation can be used to define an internal parameterization of the solid. Values will be from the local codelist for solid interpolations, see 6.4.29.

```
Solid::interpolation:SolidInterpolation[0..1]
```

#### 6.4.28.8 Attribute knot: Knot[0..3]

If necessary, 3 knot arrays will be supplied (usually only for parametric solids such as 3D b-splines).

#### 6.4.28.9 Operation: Solid (constructor)

Since this document is limited to 3-D spatial coordinate reference systems, any solid is definable by its boundary. The default constructor for a Solid is from a properly structured set of Shells organized as a solid Boundary.

```
Solid::Solid(boundary:Curve):Solid

Solid(data:SolidData):Solid
```

### 6.4.29 Datatype SolidData

The datatype SolidData (Figure 18) contains all data for constructing a solid.

#### 6.4.29.1 Attribute boundary: CurveData*[1..*]

The attribute boundary is an array of references to CurveData that will be boundary of the solid:

#### 6.4.29.2 Attribute segment: SolidData*[0..*]

The attribute segment, inherited from PrimitveData, is an array of references to SolidData that will be subsets of this solid. These solids should have disjoint interiors, and with the solid segment defined by the boundary above, the union of all these should be connected.

### 6.4.30 CodeList: SolidInterpolation

SolidInterpolation is a list of codes for 3D objects that may be used to identify the interpolation mechanisms specified by an application schema. Valid values for "interpolation" include, but are not limited, to the following:

**Table 6 — Values for SolidInterpolation**

| Value | Definition |
|---|---|
| none | The solid is solely defined by its boundary |

**Table 6** *(continued)*

| Value | Definition |
|---|---|
| linear or trilinear | Control points (some of which may be outside of the solid boundary) are organized into 3D arrays, and each cube in the array uses the unit cube in $\mathbb{E}^3$. $\{(x, y, z) \mid 0 \le x \le 1, 0 \le y \le 1, 0 \le z \le 1\}$ interpolating using trilinear interpolation. |
| tetrahedron | The space will be broken into some number of tetrahedrons, each using a local linear interpolation. Using each tetrahedron is given by 4 control points would use barycentric coordinates to parameterize the convex hull of these 4 points. Solids using this method will always be convex hulls of their corners. This in essence maps a standard 4-space Euclidean parameter space define by $\{(u_1, u_2, u_3) \mid 0 \le u_i \le 1,\ u_1 + u_2 + u_3 = 1\}$ to a standard 3-space Euclidean tetrahedron defined by the points {(0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1)} and thence to the coordinate system (regardless of dimension) of the solid being defined. This mechanism mimics the triangulated surfaces in 2d. |
| parametricCurve | The control points are organized into irregular 3-dimensional grid, each line parallel to a dimensional axis in the parameter space is associated with a curve parameterization using the appropriate controlpoints and the union of all the curves images can result in a full 3D parameterization of the space. This is similar to the parametric curve surface approach defined for surface interpolations. |
| polynomialSpline | A parametric curve interpolation where the interpolating curves are spline functions. |
| bezier | The control points are organized into irregular 3-dimensional grid, each cell within this grid is spanned by a Bézier spline function. |
| b-spline, nurbs | The control points are organized into an irregular 3-dimensional grid and each cell within this grid is spanned by a basis spline function. If the coordinates are in homogeneous format, this is a rational spline, and the associated Boolean flag "isRational" will be set to true. |

### 6.4.31 Interface Collection

#### 6.4.31.1 Semantics

A geometry collection (see Figure 19) is a type collection of geometry objects that acts as a set union of its elements. The operations add and remove act as union and difference, not necessarily as would be expected as a set of sets. Implementations wishing to implement a "set of sets" cannot use Geometry Collections (a subtype of Geometry), but could use a simple array of geometry objects.

#### 6.4.31.2 Attribute elementType: GeometryType [0..*]

The attribute elementType is a list of GeometryTypes that are allowed in this collection. The value of this attribute is normally set at construction based on the purpose for this particular instance of geometry collection.

```
Collection::elementType:GeometryType[0..*]
```

**REQ. 124** **Any element in a geometry collection shall be a type listed in the element type array, or be a subtype of a type listed.**

**Per. 3** **If the collection is to be used for a particular purpose that would limit the types listed, then the attribute elementType may be read-only.**

**REQ. 125** **Each object in the collection shall be an instantiation of one or more of the types listed in the elementType array.**

#### 6.4.31.3 Attribute numElement: Integer

The read-only attribute numElement returns the count of the elements in the collection. The only manner to change the numElement value is to add or delete elements in the collection.

```
Collection::numElement:Integer
```

### 6.4.32 Role element: Geometry

#### 6.4.32.1 The role element is an alternate access point to the elements in the Geometric object. The two logical arrays Attribute element and role: element are identical.

```
Collection::element:Geometry[0..*]
```

#### 6.4.32.2 Operation: add (geo: Geometry): Boolean

The operation "add" inserts a new element into the collection. A geometry collection acts like a set of direct positions (definition of geometry), and its "canonical representation" will union all the elements added to the collection.

```
Collection::add(geo:Geometry):Boolean
```

**REQ. 126** **The set theoretic effect of add shall be equal to the set union of this object and the object passed as the parameter geo.**

#### 6.4.32.3 Operation: remove (geo: Geometry): Boolean

The operation "remove" deletes an existing element from the collection.

```
Collection::remove(geo:Geometry):Boolean
```

**REQ. 127** **The set theoretic effect of a remove shall be equivalent to a set difference of this Collection and the object passed as geo.**

#### 6.4.32.4 Constructor: Collection (data: CollectionData): Collection

The result of the constructor that takes an array of Geometry objects is to union the input objects as a set of sets of DirectPositions.

```
Collection::Collection(data:CollectionData):Collection
```

**REQ. 128** **The CollectionData datatype contains an array of GeometryData and is the default constructor for Collection. Therefore, an array of Geometry instances in the same coordinate system shall be capable to be type cast to an instance of Collection.**

#### 6.4.32.5 Association: element: Geometry

The "element" association is an ordered array of linkages to Geometries.

**REQ. 129** **The type of any element member of the association shall be a type listed in the elementType array, or subtype of such a type.**

### 6.4.33 DataType CollectionData

The default constructor datatype is CollectionData which, in addition to the inherited attributes from GeometryData has one array of references "element" to GeometryData*[1..*].

### 6.4.34 Interface Complex

A geometric complex (Geometry Complex) is generated by a set of primitive geometric objects of the same topological dimension (in a common coordinate system) whose interiors are disjoint. Further, if a primitive is in a geometric complex (as a set of primitives), then there exists a set of primitives one dimension smaller in that complex whose union is the boundary of this first primitive.

A subcomplex of a complex is a subset of the primitives of that complex that is, in its own right, a geometric complex. A supercomplex of a complex is a superset of primitives that is also a complex. These definitions are essentially subset and superset with the added restriction that they must be a complex. A complex is maximal if it is a subcomplex of no larger complex instantiated.

The boundary of a geometric object in a geometric complex is a subcomplex of that complex. The simplest complex is a single point. The simplest 1-dimensional complex is a curve with its two ends. The simplest 2-dimensional complex is a surface with its boundary curve, and the curve's start and ends.

The underlying geometry of a complex is usually referred to as a "manifold". The structure of a complex organizes the geometry of the manifold into primitive elements, analogously to the way in which "charts" are organized by an "atlas" into a map of the world.

One way, but not the only way, to generate a complex from a set of primitives is by beginning with those primitives and performing the following Operation:

a) If two primitives overlap, then subdivide them, eliminating repetitions until there is no overlap.

b) Similarly, if a primitive is not simple, subdivide it where it intersects itself, eliminating repetitions until there is no overlap.

c) If a primitive is not a point, calculate its boundary as a collection of other primitives, using those already in the generating set if possible, and insert them into the complex.

d) Repeat steps a) through c) until no new primitive is required.

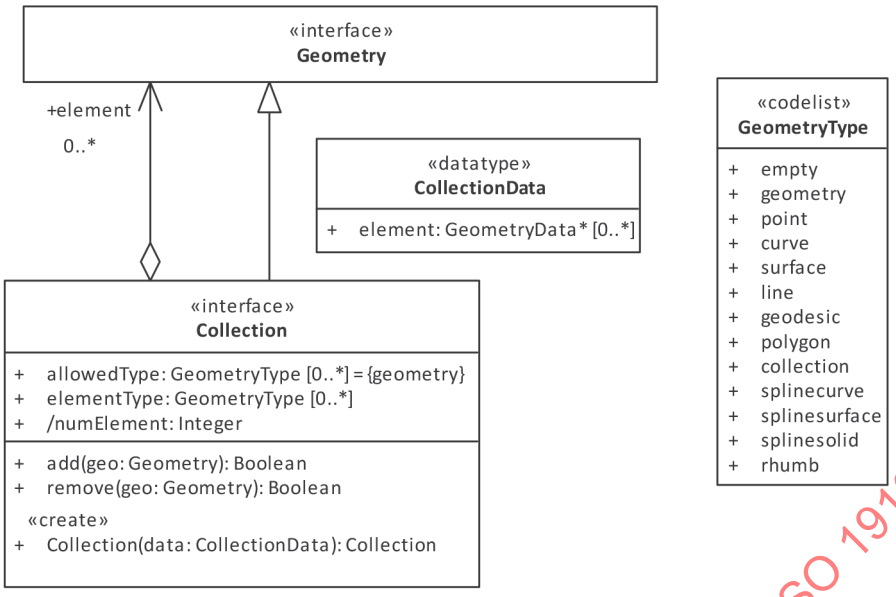



**Figure 19 — Geometry Collection**

Many systems have a concept of a universal face (for 2D) or universal solid (for 3D). This is valid only in the case where the underlying space of the complex is an unbounded Euclidean space. In this case, for 2D, the universal face is the surface in the Geometry Complex that has only interior boundary rings (its exterior one being the "point at infinity"). Analogously, in 3D, the universal solid is the one that has only interior boundary shells. In bounded manifolds, such as the sphere, there is no point at infinity, and all primitives are bounded. Without the Jordan Separation Theorem, all boundaries are essentially interior boundaries. In other unbounded manifolds, such as a hyperbolic surface, there may be more than one unbounded primitive. Since this document does not directly address these sorts of unbounded manifolds, the cardinality of some elements may require relaxing if this document were to be applied to such non-geographic manifolds. This document does not treat either the universal face or solid as special cases, and the relationship between them and their boundaries is represented in the same manner as any other boundary relationship.

A maximal complex could reasonably be considered a strong aggregation of its primitives depending on the internal semantics of the application. For this reason, the mechanism for the containment of geometry Primitives in a Geometry Complex is left unspecified. If a strong aggregation is used for maximal complexes, then the containment association for subcomplexes may have to use the maximal complex as a namespace for the references to primitives within it. In any case, once a Geometry Primitive is within a complex, or a Geometry Complex is a subcomplex of a maximal geometry Complex, its boundary operation will not need to construct representative Geometries, since by the definition of a complex, the objects needed to represent the boundary of the contained object will already exist, and only references to those objects are required by the Geometry::boundary operation. Remember that the containment of geometry Complexes in one another is a subset superset association, while the containment of Geometry Primitives in a Geometry Complex is an element set association.

A geometry Complex (Figure 20) is a collection of geometrically disjoint, simple Geometry Primitives. If a Geometry Primitive (other than a Point) is in a particular Geometry Complex, then there exists a set of primitives of lower dimension in the same complex that form the boundary of this primitive.

A geometric complex can be thought of as a set in two distinct ways. First, it is a finite set of objects (via delegation to its elements member) and, second, it is an infinite set of point values as a subtype of geometric object. The dual use of delegation and subtyping is to disambiguate the two types of set interface. To determine if a Geometry Primitive P is an element of a Geometry Complex C, call: C.element().contains (P).

A Complex, a collection of primitives, derives behaviour from the common behaviour of the elements which coincides with the behaviour of the common root, Geometry, of the Complex and its elements.

Complexes shall be used in application schemas where the sharing of geometry is important, such as in the use of computational topology. In a complex, primitives may be aggregated many to many into composites for use as attributes of features.

**REQ. 130** **A geometry complex shall contain only simple geometry primitive.**

**REQ. 131** **A geometry complex shall contain the boundary of each of its members as a set of geometry primitives.**

A geometry complex (see Figure 20) begins with a collection of primitives of the same dimension (called the "generators") with disjoint interiors, i.e. they only intersect on common boundaries. Other primitives are added so that the collection is "complete" under the boundary operator, i.e. the boundary of any primitive in the collection has its boundary representable as other primitives in the Geometry Complex.

**REQ. 132** **All direct positions contained in a geometry complex shall be interior to one and only one primitive.**

At each dimension, the set of primitives in the complex of that dimension forms the generators of a complex of that dimension. For a 3D complex, the structure works as follows:

For each dimension, n = 3, 2, 1 and 0, there exists a collection $C_n$ of geometric primitives of that dimension with disjoint interiors such that the elements of the boundary of objects in $C_n$ are in $C_{n-1}$.

$$C_3 \xrightarrow{\partial} C_2 \xrightarrow{\partial} C_1 \xrightarrow{\partial} C_0 \xrightarrow{\partial} \varnothing \tag{46}$$

G, the geometric complex is the disjoint union of the interiors of the elements of $C_0, \ldots C_n$. A geometry complex when given as topological objects is also a topological complex.

G is also the union of the generators (as geometry objects, the generators are closed).

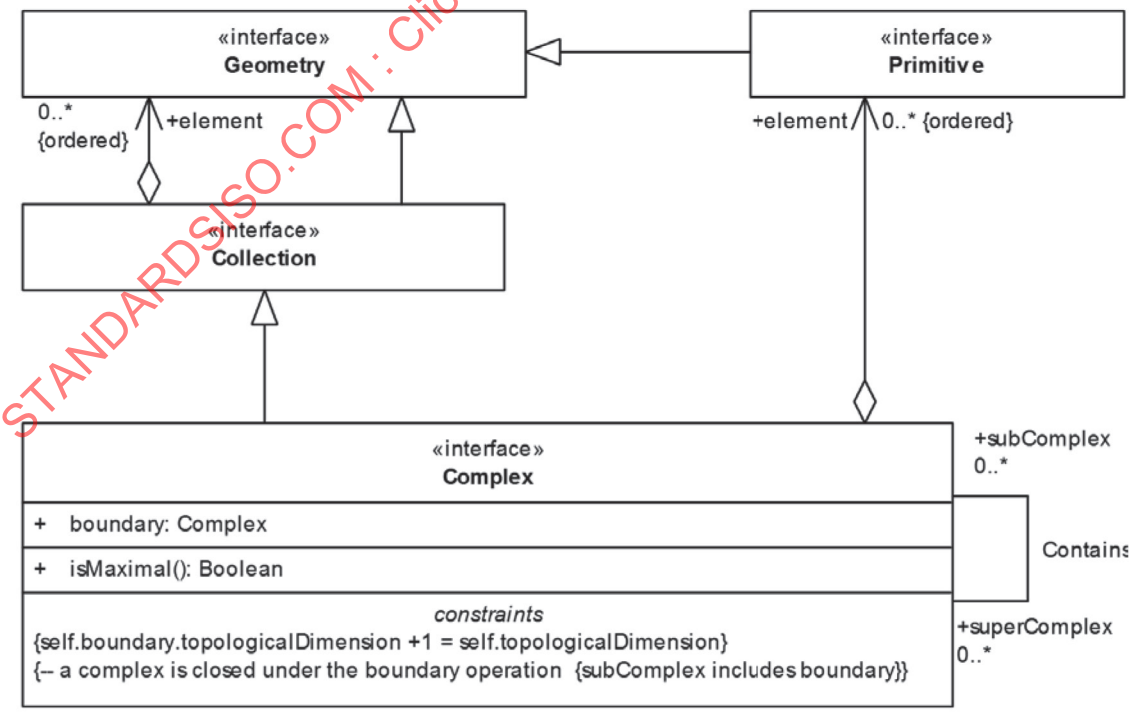



**Figure 20 — Geometric Complex**

### 6.4.35 Role Complex: generator: Primitive

The array associated with the role generator contains the primitives of the top dimension of the complex.

```
Geometry Complex::generator:Geometry Primitive [1..*]
```

### 6.4.36 Role Complex: superComplex and subComplex

A superComplex is any complex that contains all the elements of the current complex, and is hence a super set of this object, both as a geometry object and as a collection of geometry objects. A subComplex is contained in the "superComplex" target, and is a subset of it both as a geometry object and as a collection of geometry objects.

```
Complex::superComplex:Complex[0..*]

Complex::subComplex:Complex[0..*]
```

#### 6.4.36.1 Cellular complexes (informative example for use in topology)

There are standard "primitives" called cells for each dimension. In each Euclidean space, there are two standard geometries defined, a disk and a sphere:

1. $D^n = \{p \in \mathbb{E}^n \mid \|p\| \leq 1\}$ i.e., all points in $\mathbb{E}^n$ in the unit disk centred at the origin.
2. $S^{n-1} = \{p \in \mathbb{E}^n \mid \|p\| = 1\}$ i.e., all points in $\mathbb{E}^n$ on the unit sphere centred at the origin.

From a topological point of view, the boundary of the disk is the sphere: $\partial D^n = S^{n-1}$; and the sphere is the union of its "northern (positive last coordinate)" and "southern (negative last coordinate)" hemisphere, each of which is topological equivalent to a disk: $S^n \approx D^n_{south} \bigcap D^n_{north}$. By slicing the sphere into as many equal dimensioned disks as necessary, any geometric structure can be represented by collections of cells each topologically isomorphic to an appropriately dimension n disk. Thus, it turns out that all topological structures of general interest to geometry can be represented by a cellular complex in which each primitive is topological equivalent to a disk of the proper dimension.

The purpose of the definition of simple for geometry objects is to keep the decomposition in "cells" easy.

#### 6.4.36.2 Operation: Complex: isMaximal

The Boolean valued operation "isMaximal" returns TRUE if and only if this Geometry Complex is maximal that is contained in no supercomplex in the data.

```
Complex::isMaximal():Boolean
```

#### 6.4.36.3 Contains association

The association "Contains" instantiates the “contains” operation from Geometry Primitive as an association.

```
Complex::subComplex[0..n]:Geometry Complex

Complex::superComplex[0..n]:Geometry Complex
```

#### 6.4.36.4 Association role element

The association role "element" is inherited from Collection.

```
Complex::element:Primitive[0..*]
```

If a complex contains a Geometry Primitive, then it must also contain the elements of its boundary.

```
Complex:

-- closed under the boundary operation

self→forAll(self→includesAll(boundary))
```

## 6.5 Requirements Class Geometry Data

**REQ. 133** **An implementation of the requirements class Geometry Data shall support all datatypes in Requirements Class Geometry.**

# 7 Interpolations for Curves

## 7.1 Requirements Class Line Curve

### 7.1.1 Semantics

This package (see Figures 21 and 22) contains the first curve segments based on the path of shortest distance between points. In a Euclidean space such as a planar Reference Surface, this is the straight lines between points. Some arguments have been made that "line" and "geodesic" should be the same class, but the term "line" is often used to mean a "linear interpolation" between points based on the coordinate system. In most cases in geography where the "surface is curved", this differs from "geodesic" which is always the path of shortest length between nearby points.

All lines in this document are linear interpolations in the coordinate system being used.

**REQ. 134** **An implementation of the Requirements Class Line Curve shall implement Requirements Class Geometry.**

**REQ. 135** **An implementation of the Package Lines shall have all instances and properties specified for this package, its contents and its dependencies, contained in the UML model for this package in this document.**

**REQ. 136** **An implementation of the interface Line shall have all properties specified for it in the UML model in this document.**

Math In differential geometry, the geodesics (especially on surfaces in 3-space) are usually defined as ones in which the "normal" (the direction of the 2nd derivative of the curve with respect to arc length) is orthogonal to the surface. It is an exercise in the calculus of variations to show that these are the curves of shortest distance constrained by the surface.

### 7.1.2 Interface Line

#### 7.1.2.1 Semantics

A Line (Figure 23) consists of sequence of line segments, each having a parameterization between two consecutive dataPoints.

The dataPoints (inherited from Curve) of a Line are a sequence of positions between which the curve is linearly interpolated. The first position in the sequence is the startPoint of the Line, and the last point in the sequence is the endPoint of the Line.

#### 7.1.2.2 Attribute interpolation: CurveInterpolation=linear

For the interface Line, the interpolation attribute (inherited from curve) is always "linear".

```
Line::interpolation:CurveInterpolation="linear"
```

Each segment between any two of dataPoints is always a line segment in $\mathbb{R}^n$ (n is the dimension of the direct position in the point array) as a vector space. The effective interpolation between any two points (with index "i" and "i+1" in the point array) is:

$$c(\lambda)=(1-\lambda)\vec{P}_i+\lambda\vec{P}_{i+1} \text{ for } \lambda\in[0,1] \tag{47}$$

The knot array, inherited from Curve is the sequence of real numbers that mark to movement from one segment to another.

$$c(t)=\left(1-\frac{(t-k(i))}{(k(i+1)-k(i))}\right)\vec{P}_i+\left(\frac{(t-k(i))}{(k(i+1)-k(i))}\right)\vec{P}_{i+1} \text{ for } t\in[k(i),k(i+1)] \tag{48}$$

For the entire curve, the first knot is the 'startConstrParam' of the curve, and the last knot used is the 'endConstrParam' of the curve. In the above equation, a local variable $\lambda$ is defined from the constructive parameter $t$ and the values of the consecutive knots in the knot array:

$$\lambda=\frac{(t-k(i))}{(k(i+1)-k(i))} \text{ for } k(i)\le t\le k(i+1) \tag{49}$$

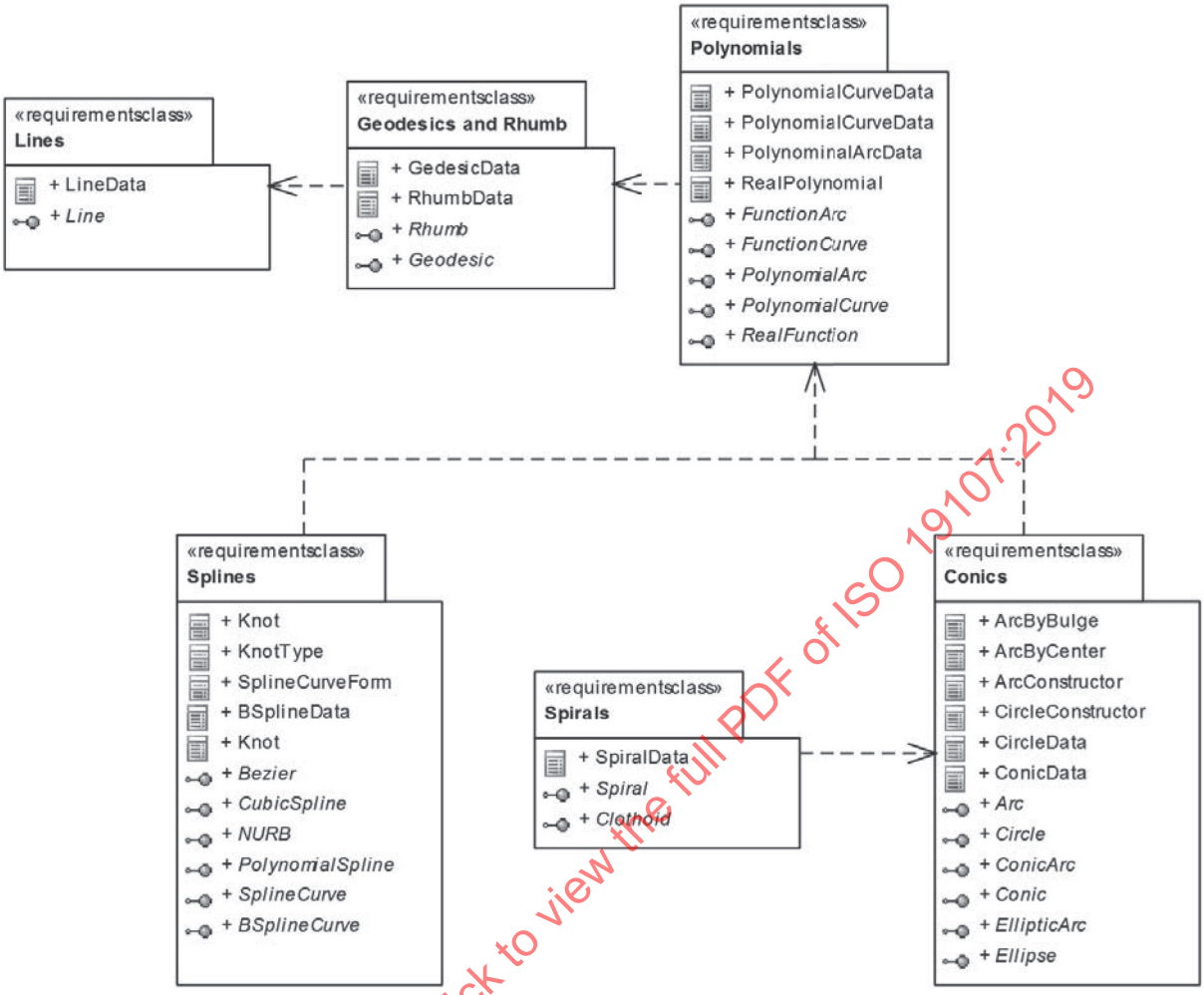



**Figure 21 — Curve Packages and Requirements Classes**

#### 7.1.2.3 Operation: Line (constructor)

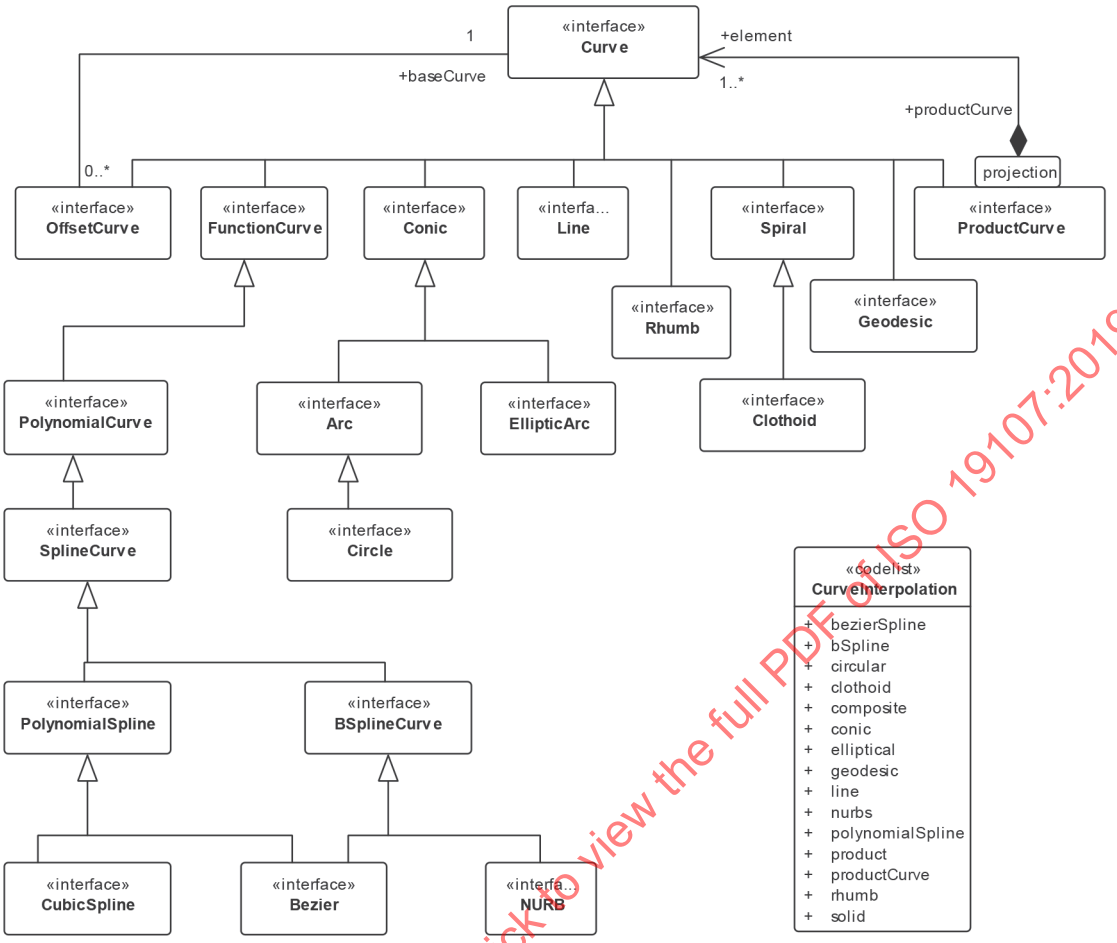
The constructor for Line takes a sequence of points and constructs a Line with those points as the controlPoints/dataPoints.

```
Line::Line(data:LineData):Line
```

### 7.1.3 DataType LineData

The default constructor for the interface Line inherits all its attributes from CurveData.

## 7.2 Requirements Class Line Data




**REQ. 137** **An implementation of the requirements class Line Data shall support all datatypes in Requirements Class Line.**

**Figure 22 — Curve Interfaces**

## 7.3 Requirements Class Geodesic Curve

### 7.3.1 Semantics

In a more general setting, where the GeometricReferenceSurface is not flat but curved (as on a geoid, ellipsoid or sphere), the shortest path between points is a geodesic. The term applies on any manifold that has an associated Gaussian or Riemannian metric (which acts as a "dot" product of the vectors in the tangent space at each point), including the situation where geometry gets is name ("geo" means Earth).

**REQ. 138** **An implementation of the Requirements Class Geodesic Curve shall implement Requirements Class Geometry.**

**REQ. 139** **An implementation of the Package Geodesics and Rhumb shall have all instances and properties specified for this package, its contents and its dependencies, contained in the UML model for this package in this document.**

On a sphere, the planes through the centre of the sphere cut the sphere in great circles. The second derivative of a circle is an inward vector pointing towards the centre of the circle and constrained by both the plane and the sphere. Therefore, the normal is orthogonal to the sphere and the great circles

are geodesics. On a Mercator projection of the sphere, the great circles are "sine wave" looking curves that pass through the equator every 180° (the intersection of the equator and the other circles are antipodal – perfect opposite positions on the sphere). On an ellipsoid, the meridian and equatorial planes cut the surface in such a manner as to be geodesics. The other geodesics (in a Mercator projection of the ellipsoid) are the same sort of "sine wave" looking curves, and they do pass through the equator, but the period of their passing is smaller than the 180° of the sphere. These geodesics are not planar in 3 dimensions.

### 7.3.2 Interface Geodesic

#### 7.3.2.1 Semantics

A Geodesic (Figure 22) consists of sequence of geodesic segments. The class is an array of geodesic segments, each segment spanning the consecutive dataPoints.

**REQ. 140 A geodesic shall represent a set of geodesic curves between data points (see 3.41).**

A geodesic is a curve on the GeometricReferenceSurface being used; since the Reference Surface is embedded in a Euclidean 3-space, $\mathbb{E}^3$, it is also a curve in 3-space. The tangent vector $\dot{c}(t)$ at a point on the surface is also a vector in $\mathbb{E}^3$. If it is normalized (made a unit vector) the result is called the "unit speed" tangent ($\vec{T} = \dot{c}(\mathrm{t}) / \left\| \dot{c}(\mathrm{t}) \right\| = \dot{c}(s)$ where the $\|\cdot\|$ is the length of the vector). The derivative of this vector with respect to arc length is the curvature vector and is normal to the unit tangent for any curve. For a geodesic, it is also normal the surface, and depends only on the surface and its local radius $\left(\kappa\vec{N}; \kappa = 1/r;\right)$. If the radius of the sphere is $r \neq 0$ then $\kappa = 1/r$. On an ellipsoid, the radius represents the best fitting sphere tangent to the geodesic and ellipsoid at that point.

#### 7.3.2.2 Attribute interpolation

```
Geodesic::interpolation:CurveInterpolation="geodesic"
```

The spatial-interpolation between any two dataPoints is a geodesic, as calculated by the associated GeometricCoordinateSystem, see 6.2.8. The basic computational requirements for this interpolation are supplied by the operations in the GeometricCoordinateSystem for "pointAtDistance" in Clause 0, "distance" in Clause Rec. 5, and "bearing" in Clause 0.

#### 7.3.2.3 Operation: Geodesic (constructor)

The constructor for Geodesic takes a sequence of points, interpolates using geodesic segments defined from the geoid (or ellipsoid) of the coordinate reference system being used, and creates the appropriate geodesic string joining them.

```
Geodesic::Geodesic(data:GeodesicData):Geodesic
```

### 7.3.3 DataType GeodesicData

The default constructor for the interface Geodesic inherits all its attributes from CurveData.

## 7.4 Requirements Class Geodesic Curve Data

**REQ. 141 An implementation of the requirements Class Geodesic Curve Data shall support all datatypes in Requirements Class Geodesic Curve.**

## 7.5 Requirements Class Rhumb

### 7.5.1 Interface Rhumb

#### 7.5.1.1 Semantics

A Rhumb line or loxodrome (Figure 23) consists of sequence of curve segments, each having a parameterization between two consecutive dataPoints where the bearing (magnetic) is constant.

The first position in the sequence is the startPoint of the curve, and the last point in the sequence is the endPoint of the curve.

**REQ. 142** **An implementation of the Requirements Class Rhumb shall implement Requirements Class Geometry.**

**REQ. 143** **A Rhumb curve, similarly to a Line, shall pass through all latitudes and longitudes in the bounding box between any two consecutive points in it dataPoint values.**

**REQ. 144** **A Rhumb shall represent a set of rhumb curves between data points (see 3.79).**

EXAMPLE As in the example after REQ. 119, A rhumb from (0, -180) to (0, 180) is defined by the rhumb line bearing between the two "coordinate points" which are in reality, the same point by REQ. 10, This Rhumb is a simple closed curve and fits the definition of a ring. The northern hemisphere is to the left. This is a valid Rhumb but also a Line and a Geodesic under the definition of geometry equals (containing the same DirectPositions).

```
Rhumb::point(≡dataPoint):DirectPosition[2..*]
```

#### 7.5.1.2 Attribute interpolation

For the interface Rhumb, the interpolation attribute (inherited from curve) is always "rhumb". The controlPoint and dataPoint arrays are identical, and may reference the same internal storage.

The knot array is the sequence of real numbers that mark to movement from one segment to another.

```
Rhumb::interpolation:CurveInterpolation="rhumb"
```

#### 7.5.1.3 Rhumb (constructor)

The constructor for Rhumb lines takes a sequence of points and constructs a sequence of loxodromic segments with those as dataPoints.

```
Rhumb::Rhumb(data:RhumbData):Rhumb
```

### 7.5.2 DataType RhumbData

The default constructor for the interface Rhumb inherits all its attributes from CurveData.



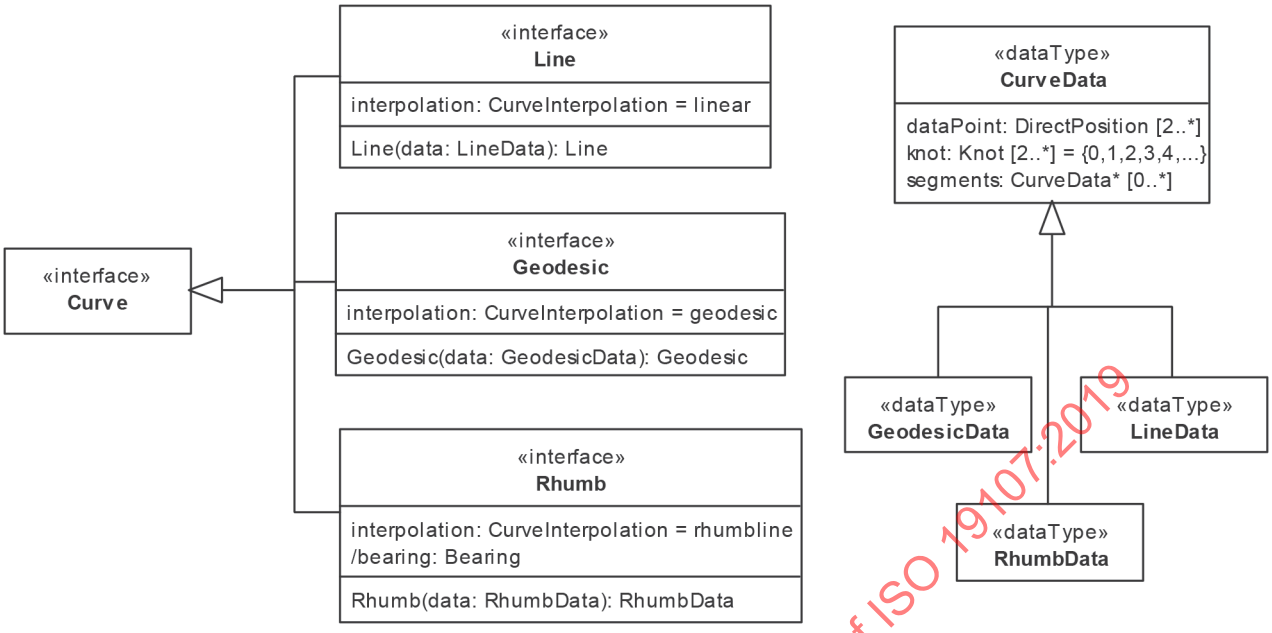



**Figure 23 — Line, geodesic and rhumb curves**

## 7.6 Requirements Class Rhumb Curve Data

**REQ. 145** **An implementation of the requirements Class Rhumb Curve Data shall support all datatypes in Requirements Class Rhumb.**

## 7.7 Requirements Class Polynomial Curves

### 7.7.1 Semantics

The formulae for Lines (0) use functions $[f_i(t)]$, usually linear polynomials, in vector equations using the coordinate for dataPoints $\left(\vec{P_i}\right)$ of the curve. The simplest generalization of this is to replace the linear coefficient functions with polynomials or other functions. This would look like this:

$$\exists f_1, f_2, \ldots, f_n \ni f_i : [a,b] \to \mathrm{R};\ c(t) = \sum_{i=1}^{n} f_i(t)\vec{P_i} \tag{50}$$

Another form would use the canonical basis for the coordinate space, i.e.

$$\vec{P_i} = \left(\delta_i^j\right),\ for\ 1 \le i \le n = dim;\ where\ \delta_i^j = \begin{cases} 0\ if\ i \ne j \\ 1\ if\ i = j \end{cases} \tag{51}$$

In which case the formula reduces to $c(t) = (f_1(t), f_2(t) \ldots, f_{\dim}(t))$.

**REQ. 146** **An implementation of the Requirements Class Polynomial Curves shall implement Requirements Class Line.**

**REQ. 147** **An implementation of the Package Polynomials shall have all instances and properties specified for this package, its contents, and its dependencies, contained in the UML model for this package in this document.**

**REQ. 148** **An implementation of the interfaces FunctionalCurve, Polynomial Arc and their dependencies shall have all properties specified for it in the UML model in this document.**

### 7.7.2 Interface RealFunction

#### 7.7.2.1 Semantics

A RealFunction is any well-defined mapping from an interval of Real numbers to the Real numbers.

#### 7.7.2.2 Attribute name

The attribute "name" of the function is a locally defined identifier for this function.

```
RealFunction::name:GenericName
```

#### 7.7.2.3 Attribute domain

The domain is the interval for which this function is defined.

```
RealFunction::domain:Interal
```

#### 7.7.2.4 Attribute metadata

```
RealFunction::metadata:URI[0..*]
```

#### 7.7.2.5 Operation: value

The operation "value" returns the value of the function for a given real in the domain.

```
RealFunction::value(r:Real):Real
```

### 7.7.3 Interface FunctionArc

#### 7.7.3.1 Semantics

The FunctionArc holds the data for one interval of the knot space for the FunctionCurve.

#### 7.7.3.2 Attribute domain

The domain is the interval for which this function arc is defined.

```
FunctionArc::domain:Interal
```

### 7.7.4 Association Role function

The parameterized association "function" is indexed by the coordinate offset, connects the arc to the real valued functions that define the coordinate offset values for the arc.



```
function(FunctionArc,coordinateOffset):RealFunction
```

### 7.7.5 Interface FunctionCurve

#### 7.7.5.1 Semantics

A FunctionCurve interface treats the coordinates as vectors in a Euclidean space. This means either that the GeometricReferenceSurface being used is planar or the global coordinate system can be locally treated as $\mathbb{E}^n$. It can be treated as a vector space or a local engineering coordinate system for an area containing the data points for the curve.

One common use is limiting the total area of any interpolation so that the scales of the various directions are more or less constant. The natural engineering space is the tangent space at the initial control point , using the basis for that space made up of the differentials for the coordinate curves [e.g. [*dx*, *dy*] for a Euclidean space, or normalize unit vectors in the directions of [*dφ*, *dλ*] for latitude (φ), longitude (λ ), or polar coordinates (distance, bearing)].

#### 7.7.5.2 Attribute FunctionCurve: numArc: Integer

The attribute "numArc" is the number of real functions needed, and thus is the number of Knot Intervals to span. It is the number of spans in the knot array (knot.length-1).

```
FunctionCurve.numArc:Integer
```

#### 7.7.5.3 Attribute FunctionCurve: metadata: URL

The attribute "metadata" is a link to a description of the function as needed.

```
FunctionCurve::metadata:URL
```

#### 7.7.5.4 Association Role segment: FunctionArc

The knotSpans are the curve segments associated to the knot intervals.

```
FuctionCurve.segment→FunctionArc[numArc]
```

### 7.7.6 Interface RealPolynomial

#### 7.7.6.1 Semantics

A RealPolynomial is real function defined by a polynomial. The complete definition of the function is given by the coefficients of terms for each degree up to the maximal degree of the particular polynomial being used. Assuming the variable is "*t*", and the coefficients are stored in an array $c_i$ for $i$ =0, 1, 2, 3... degree, then its value is given by:

$$p(t) = \sum_{i=0}^{degree} c_i (t^i) = c_0 + c_1 t + c_2 t^2 + \ldots + c_{degree} t^{degree} \tag{52}$$

#### 7.7.6.2 Attribute name, domain, metadata

The attributes "name" "domain," and "metadata" implement the same attributes from RealFunction. "Name" is a locally defined identifier for this function. Local Generic names can be represented as character strings (such as URI's) which can use fixed divider characters between names and their namespaces.

```
RealPolynomial.name:CharacterString

RealPolynomial.domain:Interval

RealPolynomial.metadata:URI[0..1]
```

#### 7.7.6.3 Attribute degree

The attribute degree is the degree of the polynomial, i.e. the largest power of the variable used.

```
RealPolynomial::degree:Integer
```

#### 7.7.6.4 Attribute c: Real

The attribute "c" is the array of the coefficients of the polynomial.

```
RealPolynomial::c[1..*]

c->length = degree + 1
```

#### 7.7.6.5 Operation: value

The operation "value" returns the value of the function for a given real in the domain.

```
RealPolynomial::value(r:Real):Real
```

The formula was given above:

$$p(\mathrm{t}) = \sum_{i=1}^{degree} c_i(t^i) = c_0 + c_1 t + c_2 t^2 + \ldots + c_{degree} t^{degree} \tag{53}$$

### 7.7.7 Interface PolynomialArc

#### 7.7.7.1 Attribute degree: Integer

The attribute "degree" is the maximum degree of all functions.

```
PolynomialArc::degree:Integer
```

#### 7.7.7.2 Constructor: PolynomialArc

The default constructor for a polynomial arc consists of the polynomial functions that define each of the offsets. When used with a PolynomialCurveData, the domain of each arc is the associated segment of the knot array.

```
PolynomialArc::PolynomialArc
(coordinateOffset:PolynomialArcData):PolynomialArc
```

#### 7.7.7.3 Association Role function (coordinateOffset: Integer): RealPolynomial

The association role "function" indicates the functions that describe each of the coordinate offsets for the arc.

```
(PolynomialArc,coordinateOffset)::function:RealPolynomial
```

### 7.7.8 Datatype PolynomialArcData

#### 7.7.8.1 Semantics

The default constructor for a PolynomialArc is the PolynomialArcData datatype. In addition to the inherited data from CurveData, it contains the coordinateOffset polynomials.

#### 7.7.8.2 Attribute CoordinateOffset

For a single arc, all that is required is the polynomial for each coordinate offsets (usually equal in number to the dimension of the coordinate system). In general, named polynomial splines derive their name from the types of constraints on the constructions, not on its final form.

```
PolynomialArcData::coordinateOffset:RealPolynomial[0..*]
```

### 7.7.9 Interface PolynomialCurve

#### 7.7.9.1 Semantics

A polynomial curve is a FunctionCurve with polynomial weights as in 7.7.1.

Most polynomial curves will be composites of simple segment arc, defined between each consecutive pair of knots in the knot space (splines, Bezier splines, B splines and, with a trick in homogeneous coordinates, NURBS). A constructor for a polynomial curve usually takes a number of constraints sufficient to create a solvable system of linear equations for the coefficients.

#### 7.7.9.2 Attribute degree

The attribute "degree" is the maximum degree of the real polynomials used.

```
PolynomialCurve::degree:Integer=(order-1)
```

Math The degree is usually one less than the number of control points. The larger the degree the more control of the curve shape the designer has. For example, a line segment has two control points, and a degree of one. In a composite, such as a spline, or NURBS, the order gives the curve designer "local control" in the sense that only the closest "order" control points contribute to a particular value. The number is usually the 1 + the degree of the polynomial functions used, the same as the number of coefficients in the polynomial. For example, a line string (degree 1) matches 2 points; a quadratic polynomial (degree 2) matches 3 points.

### 7.7.10 DataType PolynomialCurveData

The datatype PolynomialCurveData inherits from CurveData all that it needs, but the segments attribute inherited originally from Geometry is restricted to PolynomialArcs.

## 7.8 Requirements Class Polynomial Curve Data

**REQ. 149** **An implementation of the requirements class Polynomial Data shall support all datatypes in Requirements Class Polynomial.**

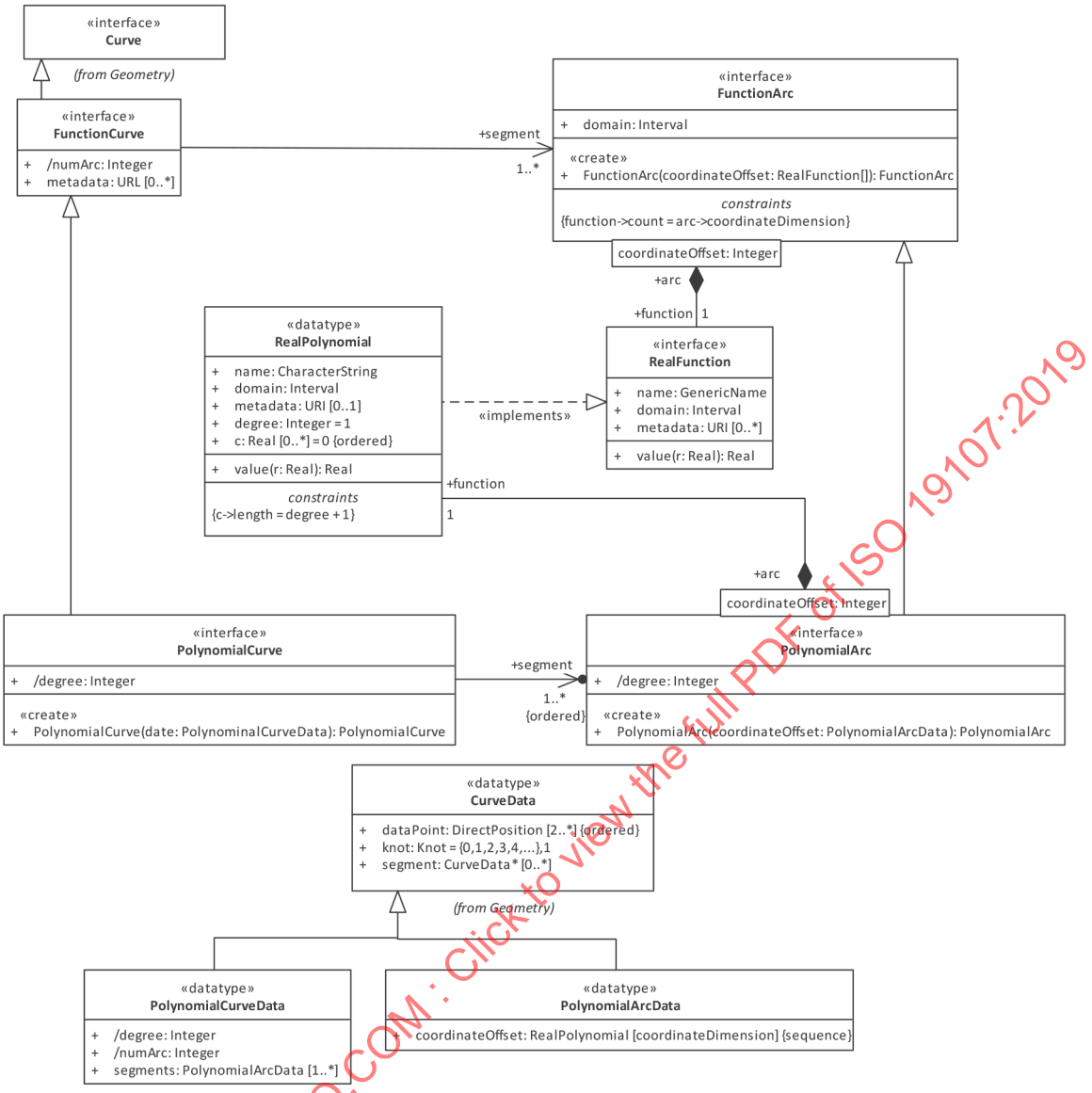



**Figure 24 — Polynomials and Polynomial Curves**



## 7.9 Requirements Class Conic Curves

### 7.9.1 Semantics

The commonality between conic and spiral curves is the use of a control point (centre or focus) and a representation for a Cartesian 2D coordinate space. The common mechanism for all curves with this property is to "draw" it in a Cartesian plane and then use a projection onto the Reference Surface. For Planes, this requires no additional work.

For the curved surfaces (spheres, ellipsoids and geoids) the easiest unifying solution is to choose an Engineering Coordinate system centred on the control point, construct the curve there and then project onto the surface. The common option is to use the tangent space at the control point, express the curve in polar coordinates (bearing and distance). Once the planar curve is defined, mapping it to the GeometricReferenceSurface used the exponential map, generates a geodesic in the given direction and map to a point along that geodesic at the given distance. This construction on any Reference Surface is called a geodesic circle.

In all the classes that follow, the basic construction of the curve will each control points and a tangent plane at that control points to construct a local segment (between consecutive data points). Once the segment is in place, the Engineering CS or tangent plane is projected onto the surface coordinates.

**REQ. 150** **An implementation of the Requirements Class Conic Curves shall implement Requirements Class Polynomial Curves.**

**REQ. 151** **An implementation of the Package Conics shall have all instances and properties specified for this package, its contents and its dependencies, contained in the UML model for this package in this document.**

**REQ. 152** **An implementation of the interfaces Arc, Circle and Conic shall have all properties specified for it in the UML model in this document.**

### 7.9.2 Interface Arc

#### 7.9.2.1 Semantics

An Arc (Figure 25) is similar to a Line except that the interpolation is by circular arcs. The controlPoints are the centres for the local arc, the dataPoint array the ends of the arcs.

#### 7.9.2.2 Attribute numArc

The attribute "numArc" will be the number of circular arcs in the string.

```
Arc.numArc:Integer
```

The $n^{th}$ arc, "`arc(n)`", is associated to the $n^{th}$ centre at "`Arc.controlPoint(n)`"and two data points, "`Arc.dataPoint(n)`" and "`Arc.dataPoin[n+1]`".

**REQ. 153** **This angular measure in radians of an arc between two consecutive dataPoints in an Arc object shall always be smaller than $\pi$.**

**REQ. 154** **If the centre is repeated, then the combined consecutive arcs associated to the same centre by value shall be less than or equal to a full circle that is 2 $\pi$ in radians.**

The number of arcs in an Arc is the length of the controlPoint array and one less than the length of the dataPoint array.

```
numArc=(controlPoint→size)=(dataPoint→size)+1
```

#### 7.9.2.3 Attribute controlPoints

The inherited controlPoint array will be used as a sequence of centres of the arcs. Beginning at the startPoint of the curve (dataPoint [1]), each controlPoint [i] will be used to centre an arc from dataPoint [i] to dataPoint [i+1]; i = 1, 2... numArc. Therefore, there will be "numArc" centres in the controlPoint array, and "numArc+1" arc endpoints in dataPoint array.

NOTE The reference system for control point in an implement can be extended to carry the radius and a vector that to two points on the circular arc: (position, radius, azimuth). If this is done, the dataPoints are defined by the "point at distance" function for the control point controlPoint[i], because radius and azimuth are essentially polar coordinates for the circle. The two unit vectors would be:

```
vector1 = radius*(sin(azimuth1), cos(azimuth1))
vector2 = radius*(sin(azimuth2), cos(azimuth2))
controlPoint[i].[pointAtDistance(vector1)=dataPoint[i]
controlPoint[i].[pointAtDistance(vector2)=dataPoint[i+1]
```

The choice of which parameters are stored and which are "derived" is an implementation decision.

#### 7.9.2.4 Attribute dataPoints

The inherited dataPoint attribute will be the array of the start and endpoints of the arcs (start and end being the shared points of two consecutive arcs in the arc string.

```
Arc:dataPoints:DirectPosition[2..*]
```

NOTE One way to visualize this it to see the dataPoints as the posts in a fence, and the controlPoints and radius anchors the data for the "rails" of the fence, in this case, the circular arcs. There is always one more post than rails.

```
dataPoint→size=controlPoints→size+1
controlPoint[i].Distance(dataPoint[i]) =
controlPoint[i].Distance(dataPoint[i+1])
```

#### 7.9.2.5 Attribute radius

Each radius[i] vector is in the tangent space of the corresponding controlPoint[i]. If the space is 3D, the radius vector, and the two endpoints of the arc will determine the plane of the arc. In all cases, the 3 points, two from the dataPoint array and the end of the vector from the control point determine the sense of the arc, starting at the first dataPoint [i], passing through the point determined by the vector (using the operation "pointAtDistance(controlPoint [i], radius [i])") and terminating at the second dataPoint [i+1]. Each arc should be less than 360° to allow for non-ambiguous sense of the arc's rotation.

```
Arc:radius:Vector[0..*]
```

#### 7.9.2.6 Constructor Arc (data:ArcData): ArcString

The only required constructor for ArcString simply supplies values for controlPoint and dataPoint arrays.

```
Arc(data:ArcData): ArcString
```

### 7.9.3 Datatype ArcData

The arc uses common supertype datatype with circle as input into the default constructor. The datatype ArcData contains:

— a list of centres as control points: `controlPoint[]:DirectPosition;`

— a list of dataPoints: `dataPoint[]:DirectPosition.`

**REQ. 155** **In ArcData, the distance from a controlPoint to two consecutive data points shall be equal; i.e. for each controlPoint:**
***controlPoint*[*i*]. *Distance*(*dataPoint*[*i*]) = *controlPoint*[*i*].*Distance*(*dataPoint*[*i* + l])**

The radian measure of any arc the first time a control point is used must be less than π radians (180°). If a larger arc is required, two arcs with the same centre can deal with anything less than 2π. In no circumstance should more than three arcs with equal centres be necessary.

Each arc will span two consecutive dataPoints, with the first shared with the previous arc and the last shared with the next. Therefore, the dataPoint array will be one longer than the controlPoint (centre) array.

The rotation of the arc will be with respect to the two radius vectors (which should not be equal nor parallel). Rotational direction is observed from the top of the cross-product vector of two consecutive radius vectors associated with the same centre point.



```
controlPoint→size+1 = dataPoint→size

radius[i].CrossProduct(radius[i+1])≠0

"Radius[i] is not colinear with radius[i+1]"
```

### 7.9.4 Interface Circle

#### 7.9.4.1 Semantics

The interface Circle is the same as that for Arc, but each arc has the same centre, the same distance between the centre (controlPoint) and the dataPoints and to be closed to form a full circle. The "start" and "end" bearing are equal and shall be the bearing for the first and last dataPoint listed.

At a minimum because the arc must be less than 360°, the controlPoint array will at least two-long with the same point in each position (the centre of the circle). The dataPoint array and the points determined by the radius vectors will give a non-ambiguous sense of the orientation of the circular curve. The "default" configuration would have *dataPoint*[0], *controlPoint*[0], and *dataPoint*[1] as points on a common geodesic diameter of the circle, starting at *dataPoint*[0] rotating towards *dataPoint*[1] in the shortest of the two directions, and then completing the circle through *dataPoint*[1], and back to *dataPoint*[0], the longer of the two arcs.

#### 7.9.4.2 Constructor Circle

The circle inherits the constructor from Arc. A 2-point ArcData would produce the circle that begins the curve continuing the direction of the first, and the shorter, of the two options.

```
Circle(a:ArcData):Circle
```

### 7.9.5 Interface Conic

#### 7.9.5.1 Semantics

The type Conic represents any general conic curve. Any of the conic section curves can be canonically represented in polar co-ordinates ($\rho$, $\theta$) as:

$$\rho = \frac{ed}{(1+e\cos\theta)} \quad for -\frac{\pi}{2} \le \theta \le +\frac{\pi}{2} \tag{54}$$

where

$e$ is the eccentricity;

$d$ is the distance to the directrix;

$P$ is semi-latus rectum.

As defined by a polynomial restriction, a general conic is determined by 5 points. This constraint can be written as the determinate of a matrix:

$$\det\begin{vmatrix} x^2 & y^2 & xy & x & y & 1 \\ x_1^2 & y_1^2 & x_1y_1 & x_1 & y_1 & 1 \\ x_2^2 & y_2^2 & x_2y_2 & x_2 & y_2 & 1 \\ x_3^2 & y_3^2 & x_3y_3 & x_3 & y_3 & 1 \\ x_4^2 & y_4^2 & x_4y_4 & x_4 & y_4 & 1 \\ x_5^2 & y_5^2 & x_5y_5 & x_5 & y_5 & 1 \end{vmatrix} = 0 \tag{55}$$

This is a general constructor that on a flat surface, a complete conic can be constructed from five dataPoints. The first controlPoint will be the centre of an exponential map that will be used to construct a conic with the first five control points, #1 to #5. The second arc will use controlPoints #5 to #9. Therefore, we can constrain the conic arcs in a single object by

```
numArc = controlPoint→size
4*numArc + 1 = dataPoing→size
```

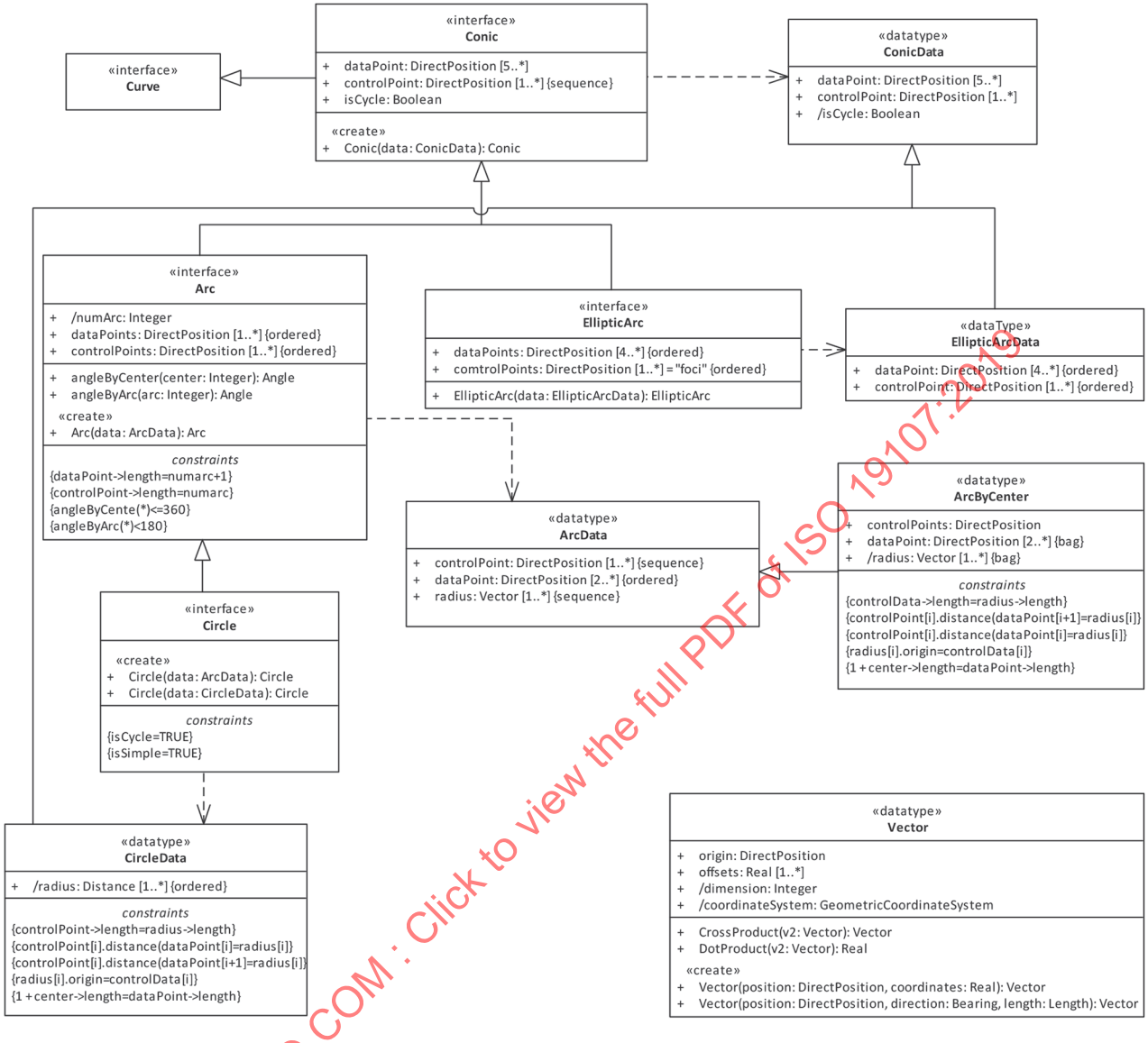



**Figure 25 — Conics, Arcs and Circles**



### 7.9.5.2 Attribute dataPoint and controlPoints

Given five distinct dataPoints, there is only one conic section that passes through them. A "cycle" can be determined by 5 points, with the startPoint and endPoint of the curve both being dataPoint[1]. If the conic is not a cycle, then we can constrain ourselves to 4 times the number of arcs plus one extra. If the isCycle=TRUE, then curve can be closed by reusing the first dataPoint as the last phantom one.

```
Conic.dataPoint:DirectPosition[5..*]

Conic.controlPoint:DirectPosition[1..*]

Conic.isCycle:Boolean
```

### 7.9.6 Interface EllipticArc, Datatype EllipticArcData

A single elliptic arc is subsegment of an ellipse, which can be determined by 4 points. The formula is similar to a generic conic (Formula 40), but only required 4 point (there is no "xy" term). The formula for each subarc given 4 points is:

$$\det\begin{vmatrix} x^2 & y^2 & x & y & 1 \\ x_1^2 & y_1^2 & x_1 & y_1 & 1 \\ x_2^2 & y_2^2 & x_2 & y_2 & 1 \\ x_3^2 & y_3^2 & x_3 & y_3 & 1 \\ x_4^2 & y_4^2 & x_4 & y_4 & 1 \end{vmatrix} = 0 \tag{56}$$

## 7.10 Requirements Class Conic Curve Data

**REQ. 156** **An implementation of the requirements class Polynomial Data shall support all datatypes in Requirements Class Conic.**

## 7.11 Requirements Class Spiral Curve

### 7.11.1 Semantics, Mathematical background: curves and curvature

Spirals are curves defined indirectly by their behaviour based on curvature, more precisely, by the local radius of curvature. This, in conjunction with speed, controls the centrifugal force that would be experienced by a physical object following the curve.

**REQ. 157** **An implementation of the Requirements Class Spiral Curve shall implement Requirements Class Conic Curves.**

**REQ. 158** **An implementation of the Package Spirals shall have all instances and properties specified for this package, its contents and its dependencies, contained in the UML model for this package in this document.**

A spiral curve is a $C^3$ curve with strictly monotonic curvature. Why this is important is traced to the formula in physics for centrifugal force:

$$a = v^2 / r \tag{57}$$

where

- $a$ is acceleration;
- $v$ is speed (giving velocity along the curve);
- $r$ is the radius of the path.

For a curve, the "$r$" is replaced by an expression of curvature:

$$a = v^2 \kappa \tag{58}$$

This means that the lateral force on a moving object (assuming a velocity) is proportional to the curvature of the path. The tighter the curve, the larger the curvature, the stronger the side force. By controlling the curvature of the road or railroad, the designer can control the side force on the vehicle (car or train) and thus control behaviour, limiting the chance of a "spinout" or "derailment".

The important fact that needs to be known is the two versions of the fundament theorem for curves. These theorems equate the definition of a curve with the definition of its curvature [$\kappa(t)$] which describes how the direction of the curve changes in time and torsion [$\tau(t)$] which describes how the curve twists (changes plane) in 3D through time. Note that these are valid only where the calculus can be used in the Cartesian coordinate space of the proper dimension (the physics is usually done in an Engineering coordinate system where the classic Newtonian forces fit the classic Newtonian equations).

For example, in a clothoid spiral, curvature is changes linear with arc length, and so with a constant acceleration (deceleration) on an opening (closing) curve, the side force is constant.

#### 7.11.1.1 Fundamental Theorem of Plane Curves (in $\mathbb{E}^2$):

Let $I$ be an interval $I=(a,b)\subseteq\mathbb{R}$ and let $\hat{\kappa}: I\subset\mathbb{R}\rightarrow\mathbb{R}$ be a continuous function. Then there exists a curve $c: I\rightarrow\mathbb{R}^2$ such that $\left\|\dot{c}(s)\right\|=1$ and the curvature of $\boldsymbol{c}$ is $\hat{\kappa}$. Any two such curves will differ by a translation and a 2D rotation.

#### 7.11.1.2 Fundamental Theorem of Space Curves (in $\mathbb{E}^3$):

Let $I$ be an interval $I=(a,b)\subseteq\mathbb{R}$ and let $\hat{\kappa}: I\rightarrow\mathbb{R}$ with $\hat{\kappa}>0$ and $\hat{\tau}: I\rightarrow\mathbb{R}$ be continuous functions. Then there is a smooth curve $c: I\rightarrow\mathbb{R}^3$ such that $\left\|\dot{c}(s)\right\|=1$ and the curvature and torsion of "$c$" are $\hat{\kappa}$ and $\hat{\tau}$. Any two such curves will differ by a translation and a 3D rotation.

#### 7.11.1.3 Using the theorems

To understand what these theorems mean and how they will be used in this document, some fundamental concepts of differential geometry need to be explained. The actual constructors of the curves in question follow the proofs of the theorems. They are direct "proof by construction".

This document uses two parameterizations for each curve interpolation type. The first is a constructive parameter ***t*** that is chosen by the implementation for their algebraic convenience. The second is arc length "***s***" which is useful in defining some differential geometric concepts. Statements in the text that do not differentiate between the two will be true for either choice.

For example, if ***c*** is a curve, then "$c(t)$" is the curve parameterized by $t$, a constructive parameter, and "$c(s)$" is the curve parameterized by s, arc length from some fixed start point, positive after that point, negative before it. To maintain the orientation of ***c***, both variables must increase in the same direction along the curve.

Because many curves are defined using standard calculus in a standard, flat, Euclidean space $\mathbb{E}2$, a local engineering CRS is required, which can then transform to the CRS of the data set. The most universally applicable Engineering CRS is the tangent plane at a particular point, which maps back to the GeometricReferenceSurface using the exponential map

Let $c(t) = (x(t), y(t))$ be a curve, which is represented by a function $c:\mathbb{R}\to\mathbb{R}^2$. The variable ***t*** is any continuous parameterization of the curve. The derivative

$$\dot{c}(t)=\left(\dot{x}(t),\dot{y}(t)\right) \tag{59}$$

is a vector in the tangent space of vectors at ***c(t)***. The length of the derivative tangent is given by the Pythagorean Theorem (***t*** and ***s*** increase in the same direction of the curve)

$$\left\|\dot{c}(t)\right\|=\left(\sqrt[2]{\dot{x}^2(t)+\dot{y}^2(t)}\right)=\frac{ds}{dt} \tag{60}$$

If ***c(t)*** is written in polar coordinates where ***r*** and ***θ*** are functions of ***t***:

$$\exists\left(r(t),\theta(t)\right)\ni c(t)=r(t)(\cos\theta(t),\sin\theta(t)) \tag{61}$$

where $r(t)=\left\|c(t)\right\|$, $x(t)=r(t)\cos\theta(t)$ and $y(t)=r(t)\sin\theta(t)$.

The unit tangent $\vec{T}$ is

$$\vec{T}(t)=\dot{c}(t)\frac{dt}{ds}=\frac{\dot{c}(t)}{\left\|\dot{c}(t)\right\|}=(\cos\theta,\sin\theta) \tag{62}$$

The vector $\vec{N}$ perpendicular to the curve, and point leftward so that $\left(\vec{T},\vec{N}\right)$ is right handed frame, is:

$$\vec{N}(t)=(-\sin\theta,\cos\theta) \tag{63}$$

Let $c(s)=\left(x(s),y(s)\right)$ be the same curve parameterized by arc length, which is also a function $c:\mathbb{R}\to\mathbb{E}^2$. The variable ***s*** is the arc length from some fixed "start point" on the curve. The derivative

$$\dot{c}(s)=\left(\dot{x}(s),\dot{y}(s)\right) \tag{64}$$

is a vector in the tangent space of vectors at $c(s)$.

Because "*s*" is arc length, $\left\|\dot{c}(s)\right\| \equiv 1$ and $\dot{c}(s)$ can be expressed in terms of the same $\theta$, where $\theta(t) = \theta(s)$:

$$\exists \theta(s) \ni \dot{c}(s) = \vec{T}(s) = (\cos\theta, \sin\theta) \tag{65}$$

Since the two parameterizations traverse the same geometry, there is a function

$$t \to s \ni c(t) = c(s) \tag{66}$$

If $\frac{dt}{ds} \neq 0$, then $\dot{c}(s) = \dot{c}(t) / \left\|\dot{c}(t)\right\| = \vec{T}(t) = \vec{T}(s)$; and similarly

$$\vec{N}(s) = \vec{N}(t) = (-\sin\theta, \cos\theta). \tag{67}$$

Since $s$ is arc length, the speed (magnitude of velocity in the direction of the tangent) is constant and the second derivative is perpendicular to the curve, i.e. perpendicular to the tangent $T$.

$$\begin{aligned} \ddot{c}(s) &= \frac{d}{ds}\dot{c}(s) \\ &= \frac{d}{ds}\vec{T}(s) \\ &= \frac{d}{ds}(\cos\theta, \sin\theta) \\ &= \frac{d\theta}{ds}(-\sin\theta, \cos\theta) \\ &= \kappa(s)\vec{N}(s) \end{aligned} \tag{68}$$

The scalar function $\kappa(s)$ is the curvature of $c$ at $c(s)$. The above formula also gives a very important connection between $\theta$ and $\kappa$:

$$\frac{d\theta(s)}{ds}=\kappa(s) \tag{69}$$

Similarly, when the normal has a similar relation:

$$\frac{d}{ds}\vec{N}(s)=\frac{d}{ds}(-\sin\theta,\cos\theta)=-(\cos\theta,\sin\theta)\frac{d\theta}{ds}=-\kappa(s)\vec{T}(s) \tag{70}$$

So, for planar curves (2D Frenet–Serret formulas):

$$\begin{aligned}\dot{T}(s)&=\kappa N\\ \dot{N}(s)&=-\kappa T\end{aligned} \tag{71}$$

In terms of $t$, the curvature $\kappa$ can be expressed:

$$\kappa(t)=\frac{\dddot{x}\ddot{y}-\dot{y}\dddot{x}}{\left(\sqrt[2]{\dot{x}^2+\dot{y}^2}\right)^3}=\frac{\pm\dot{c}(t)\times\ddot{c}(t)}{\dot{c}(t)^3} \tag{72}$$

Assuming that $\kappa(s) \neq 0$ then, the best fitting circle for the curve at $c(s)$, has a radius equal to the inverse of the curvature, $R(s) = 1/\kappa(s)$ and its centre offset from the curve in the direction $\kappa(s)\vec{N}(s)$. If the curve bends to its left, $\kappa(s) > 0$, and if it bends to the right $\kappa(s) < 0$. Constant curvature implies that the curve is either a line or a circular arc

$$\begin{aligned}&[\kappa\equiv 0]\Rightarrow[c\in Line]\\ &\left[\kappa\equiv\frac{1}{r}\neq 0\right]\Leftrightarrow[c\in Circular\ Arc\ with\ radius\ |r|]\end{aligned} \tag{73}$$

The proof in 3-space $\left(\mathbb{E}^3\right)$ is similar except that it requires solving a set of differential equations, and uses a function to describe the twisting of the curve in three space, and used the 3D version of the Frenet-Serret formulae:

$$\begin{aligned}\dot{T}(s)&=\kappa N\\ \dot{N}(s)&=-\kappa T+\tau B\\ \dot{B}(s)&=-\tau N\end{aligned} \tag{74}$$

#### 7.11.1.4 Creating a spiral from a curvature function

The relationship between curvature and arc length derived above is:

$$\frac{d\theta}{ds}=\kappa(s)$$

$$\theta(s)=\theta(0)+\int_0^s\kappa(\xi)d\xi$$

$$\dot{c}(s)=\tau(s)=(\cos\theta,\sin\theta) \tag{75}$$

$$c(s)=c(0)+\int_o^s(\cos\theta(\xi),\sin\theta(\xi))d\xi$$

A Euler spiral, clothoid or Cornu spiral is a curve whose curvature changes linearly with its curve length so there is a constant such that:

$$\exists a\in\mathbb{R}\,\text{å}\,\kappa(s)=as \tag{76}$$

Thus

$$\frac{d\theta}{ds}=\kappa(s)=as \tag{77}$$

So

$$\theta(s)=\frac{a}{2}s^2+\theta(0)=\int_0^s\kappa(\xi)d\xi \tag{78}$$

Assuming we start with the origin with a tangent the positive x-axis ($\theta = 0$):

$$\dot{c}(s) = (\cos\frac{a}{2}s^2, \sin\frac{a}{2}s^2) = \left(\cos\left(\int_0^s \kappa(\xi)d\xi\right), \sin\left(\int_0^s \kappa(\xi)d\xi\right)\right) \tag{79}$$

For the specific case of the clothoid, we can define:

$$C(s) = \int_0^s \cos\left(\frac{a\xi^2}{2}\right)\mathrm{d}\xi \text{ and } S(s) = \int_0^s \sin\left(\frac{a\xi^2}{2}\right)\mathrm{d}\xi \tag{80}$$

Then, using the Taylor series expansions for sine and cosine:

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{1+2k}}{(1+2k)!} \text{ and } \cos x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k}}{(2k)!} \tag{81}$$

gives with substitution and polynomial integration:

$$\begin{aligned} c(s) &= \int_o^s \left( \sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{a\xi}{2}\right)^{2k}}{(2k)!}, \sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{a\xi}{2}\right)^{1+2k}}{(1+2k)!} \right) d\xi \\ &= \left( \sum_{k=0}^{\infty} \frac{(-1)^k \left(\left(\frac{a}{2}\right)^{2k} s^{2k+1}\right)}{(2k+1)(2k)!}, \sum_{k=0}^{\infty} \frac{(-1)^k \left(\left(\frac{a}{2}\right)^{1+2k} s^{2(k+1)}\right)}{2(k+1)(1+2k)!} \right). \end{aligned} \tag{82}$$

This series converges quickly for small s, which is sufficient for the types of transition curves for which spirals are most commonly used.

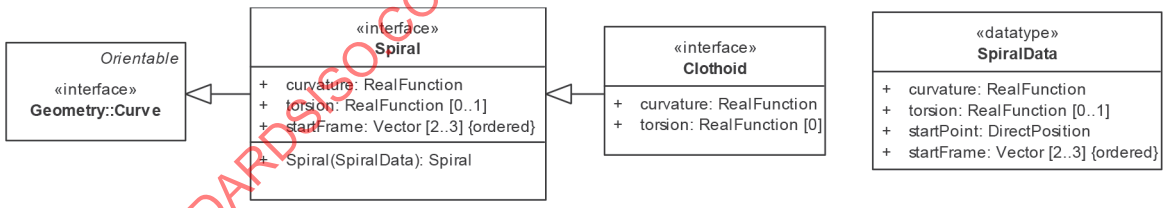



**Figure 26 — Spirals**



### 7.11.2 Interface Spiral Curves

The Spiral depends on the Frenet-Serret formulae that can create differential equations that can produce a curve based its initial position with its curvature in 2D, or with its curvature and torsion in 3D (see 7.11.1) (see Figure 26).

#### 7.11.2.1 Semantics

The general spiral is specified by its curvature function in $\mathbb{E}^2$ and by its curvature and torsion functions in $\mathbb{E}^3$. The $\mathbb{E}^n$ used is that tangent space at the start point, using an orthogonal frame of vectors based at this start point.

#### 7.11.2.2 Attribute curvature

The attribute curvature contains (or references) the curvature function of the spiral.

```
Spiral.curvature:RealFunction
```

**REQ. 159** **The curvature function shall be the curvature function of the curve in terms of its arc length from a fixed point on the infinite spiral, but not necessarily on this curve.**

**REQ. 160** **The domain of the curve knots, the first and last value in the knot array, shall be the domain of the curvature function.**

#### 7.11.2.3 Attribute torsion: RealFunction

The optional attribute torsion contains (or references) the torsion function of the spiral.

```
Spiral::torsion
```

**REQ. 161** **If the torsion function is not specified, the curve is planar and only requires curvature. If the torsion function is specified (and non-zero somewhere), the spiral shall not be planar.**

**REQ. 162** **The domain of the torsion function shall be the same as the domain of the curvature function.**

#### 7.11.2.4 Attribute startFrame

The startFrame of the curve is an orthonormal frame of two or three vectors. Each one is a unit vector, each is orthogonal to the others and together they form a right-handed frame.

```
startFrame:Vector[2..3]
```

**REQ. 163** **The startFrame shall be a set of orthogonal vectors located at the start point of the spiral.**

**REQ. 164** **If the start frame is 2D, then the curve shall be a planar spiral, with the first vector is its tangent, and the second is its normal.**

**REQ. 165** **If the start frame is 3D, then the curve is not planar, the torsion function is not everywhere zero.**

#### 7.11.2.5 Constructor Spiral

The initial conditions for the construction of a spiral are given in the datatype Spatial Data. Using the mathematics described in 7.11.1 any spiral can be defined. Extensions of this document may define subclass of Spiral by defining the form of their curvature and torsion, if needed for a non-planar curve.

```
Spiral(data:SpiralData):Spiral
```

In the next clause, an interface for clothoids is defined using its curvature function following the pattern suggested here.

### 7.11.3 Interface Clothoid Curve

The Clothoid is essentially a spiral with the torsion and curvature function defined for the entire class. The torsion function is "null" meaning that a clothoid is a planar spiral. The curvature function is linear with respect to arc length from the origin of the curve, the place where it infinite clothoid would have zero curvature. All additional data is identical to the general Spiral class.

Clothoids are used almost exclusively in road and railroad construction. Their shape works well with the physics of a vehicle following a curve. These are specialty curves, and are in a separate package because their limited use implies a separable conformance test for them.

Clothoid (Figure 26) implements the clothoid (or Cornu's spiral), which is a planar curve whose curvature is a fixed function of its length. In suitably chosen co ordinates, it is given by Fresnel's integrals:

$$x(t)=\int_0^t \cos\left(\frac{A\tau^2}{2}\right)d\tau \text{ and } y(t)=\int_0^t \sin\left(\frac{A\tau^2}{2}\right)d\tau \tag{83}$$

See Kostov[39] in the Bibliography for further properties of and methods associated with clothoid curves and piecewise clothoid curves.

This geometry is mainly used as a transition curve between curves of type straight line/circular arc or circular arc/circular arc. With this curve type, it is possible to achieve a $C^2$-continous transition between arbitrary curves. One formula for the clothoid is:

$$A^2 = R*t$$

where

A is a constant;

*R* is the varying radius of curvature along the curve;

*t* is the length along the curve and given in the Fresnel integrals.

### 7.11.4 Datatype SpiralData

SpiralData is the default constructor datatype for Spirals and contains datatypes for the four attributes of a Spiral defined in 0, 0 and 0.

**REQ. 166** **The datatype SpiralData shall contain the attributes of the Spiral interface as datatypes.**

## 7.12 Requirements Class Spiral Curve Data

**REQ. 167** **An implementation of the requirements class Spiral Data shall support all datatypes in Requirements Class Spiral Curve.**

## 7.13 Requirements Class Spline Curve

### 7.13.1 Semantics

Spline curves (see Figure 27) are mathematical approximations to the classic drafter's spline. This flexible ruler could be bent into position with pins or weights attached to it. Physically, such a device created a curve that was piecewise a cubic polynomial. Later development of splines concentrated on their mathematical and computational properties, such as how they reacted under various types of transformations, issues about local control was and how difficult they were to calculate. All splines



share the property that they can be represented by parametric functions that map (the constructive parameter of the curve) into the coordinate system of the geometric object as specified by the coordinate reference system. Spline Curves come in essentially two forms: interpolant and approximant. Both forms are examples of the FunctionCurve described in 6.4.22.

Fitted or interpolating splines ("interpolant") are exact calculations, usually for each coordinate offset separately, that create curves in each coordinate space that passes through each of the given control points. In general, the curves are defined by their dataPoints with extra conditions at boundary points (the data points at either end of the segment) and the level of continuity. For example, a cubic spline changes formula at each data point, passes through each data point, is continuous and has a smooth tangent at each point. This is the best that can be done, since a cubic polynomial only has 4 coefficients to match the 4 criteria (value at each end, and continuity of tangent at each end). The design of a cubic spline will also allow for the choice of the tangent directions at the start and end.

The second types ("approximants") only approximate the control points. These splines use sets of real valued functions that form a "partition of unity". Such functions are all defined on a single common domain, are always non-negative in their values and always sum, as a complete set, to 1,0 for their entire domain. These functions are then used in vector equations for those coordinate offset to which they will be applied, each associated with a control point, so that the tracing of the curve is a weighted average based on the partition of unity. Since the spline curve functions are all non-negative weighted sum of control points, it's value always lies in the convex hull of the control points whose weight function is currently non-zero (this is called the local convexity property), and thus always in the convex hull of the control points (the global convexity property). Since such functions are defined in vector form, they can generally be used in any target dimension coordinate system.

Most partitions of unity are originally defined as piecewise polynomials, which when applied to homogeneous coordinates and projected down to standard coordinates give us a related set of rational functions that are still a partition of unity, and hence another type of spline. Approximants have nice properties involving ease of representation, ease of calculation, smoothness, and some form of convexity. The do not usually pass through the control point, but if the control point array is dense enough, the local properties will force a good approximation of them, and will give a well-behaved curve in terms of shape and smoothness.

NOTE For polynomial curves, the system of equations to be solved will always be a linear system with variable the coefficients of the various defining polynomials, and the matrix solution depends on the constraints chosen. Given a new set of data points, the solution can be reused by replacing the old data point values with the new. This will be important in the description of tensor spline surfaces.



**REQ. 168** **An implementation of the Requirements Class Spline Curve shall implement Requirements Class Polynomial Curves.**

**REQ. 169** **An implementation of the Package Spline Curve shall have all instances and properties specified for this package, its contents and its dependencies, contained in the UML model for this package in this document.**

### 7.13.2 CodeList: KnotType

A B-spline is uniform if and only if all knots are of multiplicity one and they differ by a positive constant from the preceding knot. A B-spline is quasi uniform if and only if the knots are of multiplicity (degree+1) at the ends, of multiplicity one elsewhere and they differ by a positive constant from the preceding knot. This codelist is used to describe the distribution of knots in the parameter space of various splines. Some possible values are:

— Uniform (uniform): knots are equally space, all multiplicity 1.

— Non-uniform (nonUniform): knots have varying spacing and multiplicity.

— Quasi Uniform (quasiUniform): the interior knots are uniform, but the first and last have multiplicity one larger than the degree of the spline (p+1).

— Piecewise Bézier (piecewiseBezier): the underlying spline is formally a Bézier spline, but knot multiplicity is always the degree of the spline except at the ends where the knot degree is (p+1). Such a spline is a pure Bézier spline between its distinct knots.

Some potential values of the codelist KnotType are

```
KnotType::
                    uniform
                    nonUniform
                    quasiUniform
                    piecewiseBezier
```

This knot type is used for informational purposes, and it should be set in a manner consistent with the actual knot sequences.

### 7.13.3 CodeList: SplineCurveForm

The codelist "SplineCurveForm" is used to indicate which sort of curve is approximated by a particular spline.

Some potential values of the codelist SplineCurveForm are:

— Polyline Form, (polylineForm): a connected sequence of line segments represented by a one-degree spline (a line string).

— Circular Arc (circularArc): an arc of a circle or a complete circle.

— Elliptical Arc (ellipticalArc): an arc of an ellipse or a complete ellipse.

— Parabolic (parabolicArc): an arc of a finite subsegment of a parabola.

— Hyperbolic (hyperbolicArc): an arc of a finite length of one connected branch of a hyperbola.

```
SplineCurveForm::       polylineForm
                        circularArc
                        ellipticalArc
                        parabolicArc
                        hyperbolicArc
```

This will be used for informational purposes, and should be consistent with the other properties of the spline

### 7.13.4 Interface SplineCurve

#### 7.13.4.1 Semantics

SplineCurve (Figure 27) acts as a root for subtypes of Curves using some version of spline, either using polynomial or rational functions.



#### 7.13.4.2 Attribute curveForm

The attribute "curveForm" is used to identify particular types of curve that this spline is being used to approximate. It is for information only, used to capture the original intention. If no such approximation is intended, then the value of this attribute is NULL.

```
curveForm:SplineCurveForm[0..1]
```

#### 7.13.4.3 Attribute knot

**REQ. 170** **The attribute "knot" shall be a monotonic array of knots, each of which will define a value in the parameter space of the spline, and will be used to define the spline basis functions.**

```
knot: Knot[1..*]
```

The knot data type holds information on knot multiplicity (6.4.17). Repetitions in the knot values will be distinguished through use of this multiplicity, and so the parameter values in this array will be strictly increasing, i.e.

$$\forall i, 0 \le i < knot.length : knot[i].value < knot[i+1].value$$
$$SplineCurve :: knot : Knot\left[1..*\right] \tag{84}$$

For each knot will be associated with the corresponding data point as follows:

$$c(k_i) = dataPoint[i] \tag{85}$$

For an interpolating spline, the data points are the control points. For an approximating spline, the data points are calculated from the control points, and each data point will be generally near its corresponding control point.

#### 7.13.4.4 Attribute degree

The attribute "degree" shall be the degree of the polynomials used for defining the interpolation in this SplineCurve. Rational splines will have this degree is the limiting degree for both the numerator and denominator of the rational functions being used for the interpolation.

```
degree:Integer
```

NOTE In some sense, the multiplicity of a knot counteracts the degree of the spline in determining smoothness of a curve at the knot values, so knot multiplicity will always be less than or equal to degree. If this is not done, the curve will likely to be discontinuous unless a control point is repeated enough times.

#### 7.13.4.5 Attribute knotSpec

The attribute "knotSpec" gives the type of knot distribution used in defining this spline. This is for information only and is set according to the different construction-functions.

```
knotSpec:KnotType[0..1]
```

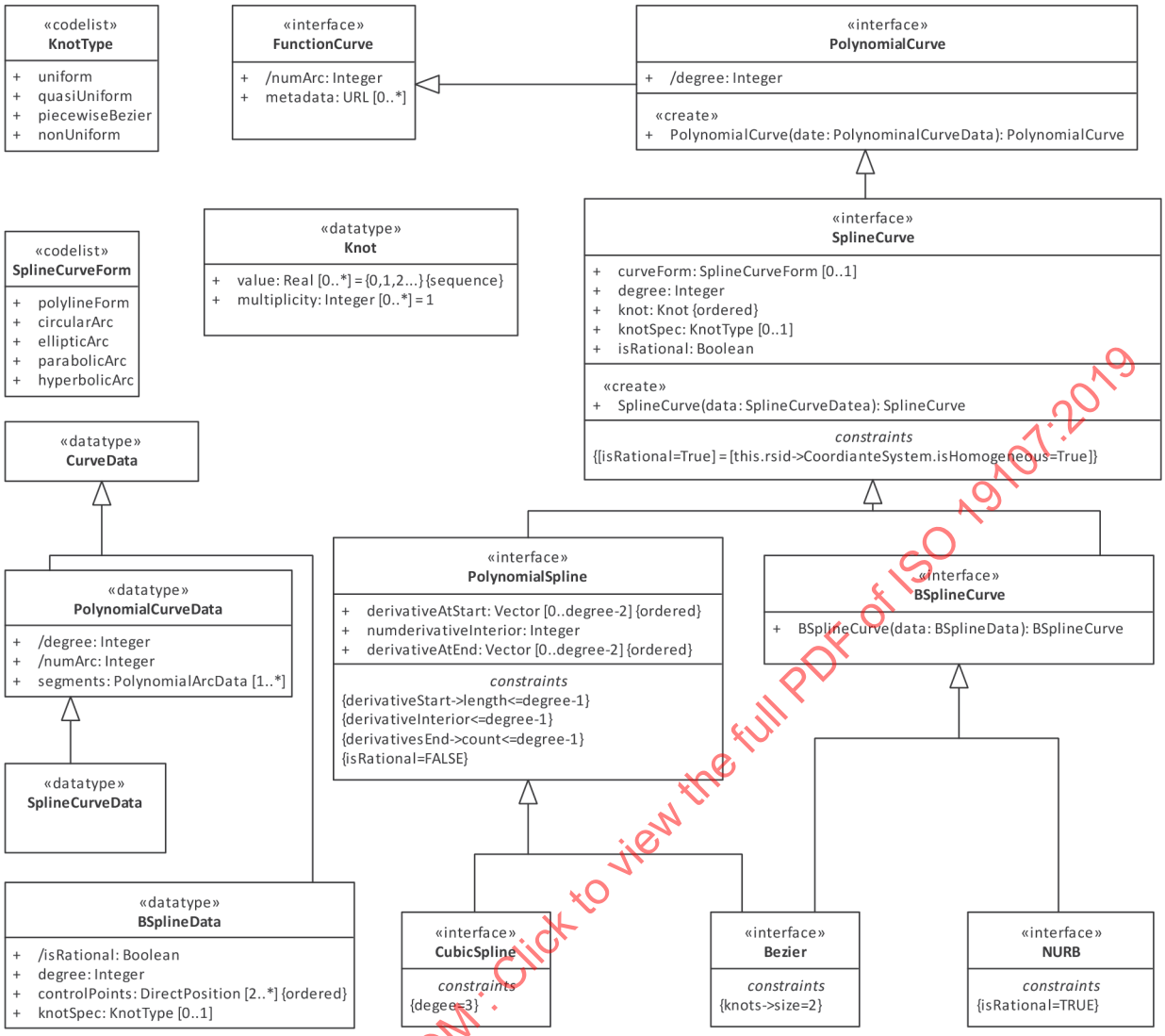



**Figure 27 — Spline Curves**



### 7.13.4.6 Attribute isRational

The attribute "isRational" indicates that the spline uses rational functions to define the curve. This is done by creating a polynomial spline on homogeneous coordinates, and projecting back to regular coordinates when all calculations are done. If the weights of all of the control points are equal, then the spline is equivalent to the corresponding polynomial spline after the projection.

`isRational:Boolean`

In a rational spline, each control point is given a weight.

$$\begin{aligned}\hat{P}_i &= (w_i x_0, w_i x_1, w_i x_2, \ldots, w_i x_{n-1}, w_i) \\ &= (x_0, x_1, x_2, \ldots, x_{n-1}, 1) \\ &\rightarrow (x_0, x_1, x_2, \ldots, x_{n-1}) = P_i\end{aligned} \tag{86}$$

If the sequence $\{\hat{P}_i\}$ contains the poles/control points use in a polynomial spline in homogeneous coordinates, then the sequence $\{P_i\}$ contains the poles/control points in the corresponding rational spline, using the same "divide by w" projection from homogeneous coordinates to regular coordinates.

Note that the denominator of the rational function is weight coefficient of the homogeneous slot in the original polynomial spline.

$$\begin{aligned}\hat{c}(u) &= \sum_{i=0}^{n-1} b_i(\mathrm{u})\hat{P}_i \\ &= \sum_{i=0}^{n-1} b_i(\mathrm{u})(w_i P_i, w_i) \\ &= \sum_{i=0}^{n-1} w_i b_i(\mathrm{u})(P_i, 1) \\ c(u) &= \sum_{i=0}^{n-1} w_i b_i(\mathrm{u}) P_i \Big/ \sum_{i=0}^{n-1} w_i b_i(\mathrm{u})\end{aligned} \tag{87}$$

**REQ. 171** **It shall be the case that "isRational" is "TRUE" if and only if the control points of the spline are in homogeneous coordinates, each point having a weight.**

$$[\text{isRational}] \Leftrightarrow [\text{controlPoint.ReferenceSystem.instanceOf.HomogeneousReferemceSystem}] \tag{88}$$

### 7.13.5 Interface PolynomialSpline

#### 7.13.5.1 Semantics

A polynomial spline is an interpolation of dataPoints, i.e. a polynomial curve passing through data points. Construction of such a spline depends on the constraints: which may include:

— on values or derivatives of the spline at the data points;

— on the continuity of various derivatives at chosen points;

— degree of the polynomial in use.

**REQ. 172** **An "$n^{th}$ degree" polynomial spline shall be defined for each direct position offset, piecewise between knot parameter values, as an n-degree polynomial, with up to $C^{n-1}$ continuity at the control points where the defining polynomial may change.**

**REQ. 173** **This level of continuity shall be controlled by the attribute numDerivativeInterior, which shall default to (degree-1).**

**REQ. 174** **Constructive parameters may include constraints for as many as "degree – 1" derivatives of the polynomials at each knot.**

NOTE A line sting is a degree 1 spline and is $C_0$ in the sense it is continuous. An $n^{th}$ degree spline can be $C_{n-t}$ meaning it can have continuous derivative up to "n–1". Therefore a cubic spline can be continuous, have a continuous slope (derivative) and curvature that depends on the second derivative.

Line is a first degree polynomial spline. For line strings viewed as splines, it can be represented as almost any type, Bézier, B Spline or Polynomial, since with "degree = 1", the formulae for each of these types collapse to a line string. Line strings have simple continuity at the controlPoints ($C^0$), but do not require derivative information (degree – 2 = –1).

The process for setting up the polynomial for each offset for a set of direct positions is solving a set of equations for the coefficients of the polynomial between each pair of sequential knots (note that for polynomial splines, all knots are degree 1):

$$\begin{aligned} &\text{knots: } k_i = [u_i, 1], \\ &\text{dataPoints: } \vec{P}_i = (x_{i,0}, x_{i,1}, x_{i,2} \ldots), \\ &c(u) = (c_o(u), c_1(u), c_2(u), \ldots), \\ &c_j \text{ is a polynomtial on } [u_i, u_{i+1}] \\ &c_j(u_i) = x_{i,j} \\ &\frac{d}{du} c_j^-(u_i) = \frac{d}{du} c_j^+(u_i) \end{aligned} \tag{89}$$

NOTE The major difference between the polynomial splines, the b splines (basis splines) and Bézier splines is that polynomial splines pass through their control points, making the control point and sample point array identical. Polynomial splines are essentially calculated by brute force (albeit sometimes "organized" brute force) from the knots, the control points, and other parameters. Because of this, they do not generalize to surface splines, except in limited cases

#### 7.13.5.2 Attribute derivativeAtStart

The attribute "derivativeAtStart" shall be the values used for the initial derivative (up to degree –2) used for interpolation in this PolynomialSpline at the start point of the spline. The attribute "derivativeAtEnd" shall be the values used for the final derivative (up to degree –2) used for interpolation in this PolynomialSpline at the start point of the spline.

```
PolynomialSpline::derivativeAtStart:Vector[0..*]                {size ≤ degree - 2}

PolynomialSpline::derivativeAtEnd:Vector[0..*]                  {size ≤ degree - 2}
```

### 7.13.6 Interface CubicSpline

Cubic splines are similar to line strings in that they are a sequence of segments each with its own defining cubic polynomial splines. The usual constraints are:

$$\begin{aligned} &\vec{c}(u_i) = \vec{P}_i \\ &\dot{\vec{c}}^-(u_i) = \dot{\vec{c}}^+(u_i) \text{ for i=1,2...,n-2} \\ &\dot{\vec{c}}(u_0) = vectorAtStart[0] \\ &\dot{\vec{c}}(u_{n-1}) = vectorAtEnd[0] \end{aligned} \tag{90}$$

A cubic spline uses the dataPoints and a set of derivative parameters to define a piecewise third degree polynomial interpolation in each coordinate dimension. Because the dimensions are handled separately, homogeneous coordinates shall not be used in this curve type. This is true for any fitted curve regardless of polynomial degree. Unlike line-strings, the parameterization by arc length is not necessarily still a polynomial. Splines have two parameterizations that are used in this document, the defining one (constructive parameter) and the one that has been parameterized by arc length to satisfy the requirements in Curve. In a CubicSpline, the constructive one is a set of cubic polynomials, on for each dimension offset in the DirectPosition used.

The function describing the curve must be $C^2$, i.e. have a continuous first and second derivative at all points and pass through the controlPoints in the order given. Between any two consecutive control points, the curve segment is defined by cubic polynomials, one for each offset in the coordinates. At each control point, the polynomial changes in such a manner that the first and second derivative vectors are the same from either side. The control parameters record must contain derivativeAtStartvectorAtStart, and derivativeAtEndvectorAtEnd which are the tangent vectors at controlPoint[1] and controlPoint [n]

where n = controlPoint.count. Since the constructive parameterization is not the arc length one, these tangent vectors may well not be unit vectors. The length of the vectors affects the shape of the curve.

The restriction on "derivativeAtStartvectorAtStart" and "derivativeAtEnd" reduce these sequences to a single tangent vector each.

```
CubicSpline.derivativeAtStart: Vector \\ "degree - 2" is 1

CubicSpline.derivativeAtEnd: Vector \\ "degree - 2" is 1
```

NOTE The actual implementation of the cubic polynomials varies, but the curve generated is guaranteed to be unique. The references [14], [22], [24], [30] and [31] in the Bibliography contain examples of implementations.

The interpolation mechanism for a CubicSpline is "cubicSpline".

```
CubicSpline::interpolation:InterpolationMethod="cubicSpline"
```

The degree for a CubicSpline is "3".

```
CubicSpline::degree: Integer= 3
```

### 7.13.7 Interface Bezier

The Bezier are approximating splines that use Bézier or Bernstein polynomials as a partition of unity. An n+1 long control point array will create a polynomial curve of degree “n” that defines the entire segment. These curves are defined in terms of the set of basis functions given by:

$$J_{n,i}(u)=\binom{n}{i}u^i(1-u)^{n-i} \text{ where } \binom{n}{i}=\frac{n!}{i!(n-i)!} \text{ for n=0,1,2,3,...} \tag{91}$$

These functions are a "Partition on Unity" which can be seen by using the Binomial theorem for degree n on $(x+y)^n$ where x=u and y=1-u:

$$1=1^n=(u+(1-u))^n=\sum_{i=0}^{n}\binom{n}{i}u^i(1-u)^{n-i}=\sum_{i=0}^{n}J_{n,i}(u) \tag{92}$$

The only knots for a Bézier are "0" and "1" and they are multiplicity "n–1". The set of "n+1" control points $\vec{P}_0$, $\vec{P}_1$ ... $\vec{P}_n$ shall determine a curve segment given by:

$$\vec{c}(u)=\sum_{i=0}^{n}J_{n,i}(u)\vec{P}_i \text{ for } u\in[0,1] \tag{93}$$

The sample points of this segment are the values of the curve defined at the maximum of each of the polynomials (i/n):

$$\forall\, i\in\{0,\ 1,\ 2\ \ldots\ n\}:\vec{S}_i=\vec{c}\left(\frac{i}{n}\right) \tag{94}$$

The only control points that the curve is forced to go through are the first and the last one in the array.

NOTE For n = 1, the two weight functions are as follows:

$$J_{1,0}(t)=\binom{1}{0}t^0(1-t)^1=(1-t) \text{ and } J_{1,1}(t)=\binom{1}{1}t^1(1-t)^{1-1}=t \quad (95)$$

Given $P_0$ and $P_1$, the curve segment becomes:

$$\vec{c}(t)=(1-t)P_0+tP_1 \text{ for } t\in[0,1] \quad (96)$$

i.e. for $n = 1$, the Bézier polynomial is geometrically equivalent to a simple line segment.

**REQ. 175** **If "c" is a Bezier with a uniform knot sequence, c.knotType="piecewiseBezier" and of degree n, then the controlPoint sequence shall have length one greater than some integer multiple n; that is, there shall exist a positive integer "s" such that "c.controlPoint.length=s*c.degree+1".**

Further, for each subsequence of control points starting at an index of a multiple of n, of length n+1, the curve will be a Bézier spline of degree "n" starting at the first point in that subsequence and ending at the last point, with that subsequence of n+1 point as poles for that part of the curve. This curve will be continuous in all cases since the last pole of each Bézier curve is the first pole of the next. It will be C1 (continuous first derivative) if the difference vector of the last two poles of one sequence is equal to the difference vector of the first two of the next sequence, i.e. if:

$$[\vec{P}_{i*n}-\vec{P}_{i*n-1}=\vec{P}_{i*n+1}-\vec{P}_{i*n}]\Rightarrow[c' \text{ is contiuous at } \vec{P}_{i*n}] \quad (97)$$

### 7.13.8 Interface BSplineCurve (and NURBS)

#### 7.13.8.1 Semantics

A B-spline curve (Figure 27) is a piecewise parametric polynomial or rational curve described in terms of control points and basis functions. If the control points are not homogeneous form or they are but the weights are all equal to one another, then it is a piecewise polynomial function. Otherwise, it is a rational function spline. The steps in the construction of such a curve follow these steps:

From $\mathbb{R}$, choose a sequence of knot values $0 \le u_0 \le u_1 \le \ldots \le u_m \le 1$ with the $i^{th}$ knot span defined by:

$$[\boldsymbol{u_i}, \boldsymbol{u_{i+1}})=\{\boldsymbol{u} \mid \boldsymbol{u_i} \le \boldsymbol{u} < \boldsymbol{u_{i+1}}\}; \boldsymbol{U}=\{\boldsymbol{u_0}, \boldsymbol{u_1}, \boldsymbol{u_2}, \boldsymbol{u_3}, \ldots, \boldsymbol{u_{n+p+1}}\} \quad (98)$$

From the CRS, choose the control points $P_i \in DirectPosition, i \in \mathbb{Z} \ni 0 \le i \le n$.

If the b-spline is to be polynomial, then the points are in a standard coordinate system and look like:

$$P_i=(x_{i,1}, x_{i,2}, x_{i,3}, \ldots, x_{i,d}) \in \mathbb{R}^n \quad (99)$$

If the b-spline is to be rational, then the points are in a homogeneous coordinate system and look like: $P_i=(w_i x_{i,1}, w_i x_{i,2}, w_i x_{i,3}, \ldots, w_i x_{i,d}, w_i) \in \mathbb{R}^{n,1}$.

From the knots, basis functions are defined recursively:

$$N_{i,0}(u)=\begin{cases}1 & u\in[u_i, u_{i+1})\\ 0 & u\notin[u_i, u_{i+1})\end{cases}$$
$$N_{i,p}(u)=\frac{u-u_i}{u_{i+p}-u_i}N_{i,p-1}(u)+\frac{u_{i+p+1}-u}{u_{i+p+1}-u_{i+1}}N_{i+1,p-1}(u) \text{ for } p>0 \quad (100)$$

A B-spline curve is a piecewise Bézier curve if it is quasi uniform except that the interior knots have multiplicity "degree" rather than having multiplicity one. In this subtype the knot spacing shall be



1,0, starting at 0,0. A piecewise Bézier curve that has only two knots, 0,0, and 1,0, each of multiplicity (degree+1), is equivalent to a simple Bézier curve.

#### 7.13.8.2 Operation: BSplineCurve

The class constructor "BSplineCurve" uses the pertinent information described in the attributes above and constructs a B spline curve. If the knotSpec is not present, then the knotType is uniform and the knots are evenly spaced, and except for the first and last have multiplicity = 1. At the ends the knots are of multiplicity = degree+1. If the knotType is uniform, they need not be specified.

```
BSplineCurve(data BsplineData):BSplineCurve
```

NOTE If the B-spline curve is uniform and degree = 1, the B spline is equivalent to a polyline (Line). If the knotType is "piecewiseBezier", then the knots are defaulted so that they are evenly spaced, and except for the first and last have multiplicity equal to degree. At the ends the knots are of multiplicity = degree+1.

### 7.13.9 DataType BsplineData

The datatype BsplineData contains 4 attribute extending the CurveData which it is a subclass:

— isRational: Boolean (true if and only the coordinate space is homogeneous);

— degree: Integer;

— controlPoints: DirectPosition[2,*] used in construction of the basis functions;

— knotSpace: KnotType a description of the type of knotSpace being used.

## 7.14 Requirements Class Spline Curve Data

**REQ. 176** **An implementation of the requirements class Spline Curve Data shall support all datatypes in Requirements Class Spline Curve.**

# 8 Interpolations for Surfaces

## 8.1 Requirements Class Polygon Surface

### 8.1.1 Semantics

A polygon is a surface defined solely its boundary curves and a spanning surface.

**REQ. 177** **An implementation of the Requirements Class Polygon Curve shall implement Requirements Class Geometry and at least one Curve Requirements Class.**

**REQ. 178** **An implementation of the Package Polygon shall have all instances and properties specified for this package, its contents and its dependencies, contained in the UML model for this package in this document.**

### 8.1.2 Interface Polygon

#### 8.1.2.1 Semantics

A Polygon (Figure 28) is a surface that is defined by a set of boundary curves and an underlying surface to which these curves adhere. The default logic has been that a polygon was "planer" and the underlying coordinate system was therefore Euclidean. In general, any topological 2D surface with a local coordinate system will work.

Care must be taken not to let the intuition of the infinite plane to color our view of a finite spheroid $S^2$ a topological 2D sphere in 3D space, see the definition of n-sphere.

In 2D spaces, the default is that the curves are on the GeometricReferenceSurface (usually the datum). If this surface is a plane the concepts of interior and exterior rings are needed, but otherwise, the only requirement is that all rings have the surface on their left (in terms of the traversal, through the dataPoints of the curve).

**REQ. 179** **If the underlying GeometricReferenceSurface is bounded and closed then the distinction between interior and exterior boundary rings is ignored.**

NOTE If the spanning surface is a GriddedSurface, then the boundary curves can use the parametric coordinates of the surface in lieu of a more standard CRS.

#### 8.1.2.2 Attribute Polygon: boundary: Curve [1..*]

The attribute "boundary" stores the surface boundary cures that are the boundary of this Polygon.

```
Polygon::boundary:Curve[1..*]
```

**REQ. 180** **The Curves in a Polygon boundary shall always be oriented so that their left side is toward the interior of the Polygon.**

**REQ. 181** **A test for determining if a Point, not on the boundary of a Polygon, is interior to the Polygon shall be that a curve from the Point that does not cross the Boundary Curves but touches one, will always arrive from the left side of the Boundary Curve.**

**REQ. 182** **If the underlying Geometric reference system is unbounded, then the order of the Curves in a Polygon boundary shall have the largest ring (the exterior ring, as measured by the MBR of the ring) first, followed by the smaller rings in any order.**

#### 8.1.2.3 Association role: spanningSurface: GriddedSurface [0..1]

The optional spanning surface provides a mechanism for spanning the interior of the polygon as a surface in 3D.

```
Polygon::spanningSurface:GriddedSurface[0..1]
```

The spanning surface should have no boundary components that intersect the boundary of the polygon, and there should be no ambiguity as to which portion of the surface is described by the bounding curves for the polygon. A common spanning surface is an elevation model, which is not directly described in this document, although Tins and ParametricCurveSurface are often used in this role. Any gridded surface can easily be used. In this case, the CRS of the Polygon can be the local coordinates in $\mathbb{R}^2$ of the gridded surface parameter space.

#### 8.1.2.4 Operation: Polygon (constructor)

This first variant of a constructor of Polygon creates a Polygon directly from a set of boundary curves that are defined using DirectPositions on the GeometricReferenceSurface as dataPoints of the curve.

```
Polygon::Polygon(boundary:Curve[1..*]):Polygon
```

NOTE 1 The meaning of "exterior ring" is consistent only if the spanning surface of the constructed planar polygon is unbounded. The term comes from the Jordan Curve Theorem on a plane. When the underlying surface is a sphere, if we construct a buffer zone of all points within 10 kilometers of the equator, there is no way to determine which side of that polygon (north or south) is exterior or interior.



This second variant of a constructor of Polygon creates a Polygon lying on a spanning surface. There is no restriction of the types of interpolation used by the composite curves used in the surface boundary, but they must all be lie on the "spanningSurface" for the process to succeed.

```
Polygon(boundary:Curve[1..*], spanSurf:Surface):Polygon
```

NOTE 2 It is important that the boundary components be oriented properly for this to work. If it is the case of a bounded manifold, such as a sphere, there is an ambiguity unless the orientation is properly used. If the spanning surface is a gridded or parametric surface, then the CRS of the curves can be the parameter space of the surface.

**REQ. 183** **An implementation of the interfaces Polyhedral Surface and Polygon shall have all properties specified for it in the UML model in this document.**

### 8.1.3 Datatype PolygonData

The datatype PolygonData (Figure 28) contains the boundary curves that will be the component boundaries of the resultant Polygon.

**Rec. 10** **Each resultant curve from the list of the PolygonData should be a cycle, but if it is not, the constructor should repeat the first point to close the curve.**

### 8.1.4 Interface PolyhedralSurface

#### 8.1.4.1 Semantics

A PolyhedralSurface (Figure 28) is a Surface composed of Polygons connected along their common boundary curves. This differs from Surface only in the restriction on the types of surface segments acceptable. The interface is subtyped under Collection, with the major differences being the PolyhedralSurface contains only polygons.

#### 8.1.4.2 Operation: PolyhedralSurface (constructor)

The constructor for a PolyhedralSurface takes the facet Polygons and creates the necessary aggregate surface.

```
PolyhedralSurfac(data:PolyhedralSurfaceData):PolyhedralSurface

PolyhedralSurace(geo:Polygon[1..*]):PolyhedralSurface
```

#### 8.1.4.3 Association role: segment: Polygon [1..*]

The association role "segment" associates this surface with its individual facet polygons. It shall be non-empty. This role is inherited from Primitive.

```
segment:Polygon[1..*]
```

### 8.1.5 Datatype PolyhedralSurfaceData

The default constructor datatype PolyhedralSurfaceData for a PolyhedralSurface (Figure 28) contains the PolygonData for each of its segment Polygons.

### 8.1.6 Interface Triangle

A Triangle is a planar Polygon, with linear edges, defined by three corners, i.e. a Triangle would be the result of a constructor of the form:

```
Triangle(Line(P1, P2, P3, P1))=Polygon(Line(P1, P2, P3, P1))
```

or

```
Triangle(P1, P2, P3)
```

where P1, P2, and P3 are three DirectPositions. The repeated closing point in the Triangle constructor is optional. Triangles have no holes. Triangle may be used to construct TriangulatedSurfaces.

**Rec. 11** **The resultant curve from the list passed to the Triangle constructors should be a cycle, but if it is not, the constructor should repeat the first point to close the curve.**

NOTE The points in a triangle in the coordinate system can be located in terms of their corner points by defining a set of barycentric coordinates, three nonnegative numbers $c_1$, $c_2$, and $c_3$ such that $c_1 + c_2 + c_3 = 1{,}0$. Then, each point $P$ in the triangle can be expressed for some set of barycentric coordinates as:

$$P = c_1 P_1 + c_2 P_2 + c_3 P_3 \tag{101}$$

### 8.1.7 Datatype TriangleData

The datatype TriangleData (Figure 28) contains the boundary curve that is the boundary of the triangle. The triangle is the convex hull of the boundary in the coordinates when treated as a Cartesian coordinate system.

### 8.1.8 Interface TriangulatedSurface

A TriangulatedSurface (Figure 28) is a PolyhedralSurface that is composed only of triangles (Triangle). There is no restriction on how the triangulation is derived. Unlike a collection of points that can be subjected to a Delaunay triangulation, the default interface for a triangular surface is the surface following the same pattern as a polyhedral surface.

Implementations that store only the points and use a specific triangulation technique to create the surface may be considered satisfying the requirements of a triangular surface because they support the interface.

### 8.1.9 Datatype TriangulatedSurfaceData

The default constructor datatype TriangulatedSurfaceData for a TriangulatedSurface (Figure 28) contains the TriangleData for each of its segment Triangles.

## 8.2 Requirements Class Polygon Surface Data

**REQ. 184** **An implementation of the requirements class Polygon Data shall support all datatypes in Requirements Class Polygon.**

## 8.3 Requirements Class Parametric Curve Surface

### 8.3.1 Semantics

A gridded or parametric curve surface is one that by the nature of its construction has a natural subdivision of its area usually by its parameterization.

**REQ. 185** **An implementation of the Requirements Class Parametric Curve Surface shall implement Requirements Class Geometry and at least one Curve Requirements Class.**

**REQ. 186** **An implementation of the Gridded Surfaces shall have all instances and properties specified for this package, its contents and its dependencies, contained in the UML model for this package in this document.**

### 8.3.2 Interface ParametricCurveSurface

#### 8.3.2.1 Semantics

The parametric curve surfaces are continuous families of curves, given by a constructive function of the form:

$$S(u,v):[a,b]\otimes[c,d]\rightarrow DirectPosition \tag{102}$$

A one-parameter family of curves can be defined by fixing the value of either parameter; fixing **v** gives a set of curves in **u**, fixing u gives a set of curves in **v**.

$$c_v(u)=c_u(v)=S(u,v) \tag{103}$$

The functions on ParametricCurveSurface (Figure 29) shall expose these two families of curves. The first gives us the "horizontal" cross sections $c_v(u)$, the later the "vertical" cross sections $c_u(v)$. The terms "horizontal" and "vertical" refer to the parameter space and need not be either horizontal or vertical in the coordinate reference system. Table 7 lists some possible pairs of types for these surface curves (other representations of these same surfaces are possible). The two partial derivatives of the surface parameterization, $\vec{i}$ and $\vec{j}$ are given by:

$$\vec{i}\equiv\frac{\partial}{\partial u}S(u,v)=\frac{d}{du}c_v(u) \quad \text{and} \quad \vec{j}\equiv\frac{\partial}{\partial v}S(u,v)=\frac{d}{dv}c_u(v) \tag{104}$$

The default upNormal ($\vec{n}$) for the surface is the normalized vector cross product of these two curve derivatives when they are both non-zero: $\vec{n}=\left(\vec{i}\times\vec{j}\right)\Big/\left\|\vec{i}\times\vec{j}\right\|$. The two vectors $\vec{i}$ and $\vec{j}$ are the images of unit, orthogonal vectors in the parameter $(u,v)$ -space of the surface but the mapping $S(u,v)$ may have a zero partial derivative at some points, in which case the curves $c_v(u)$ and $c_u(v)$ will need to be reparametrized by the local arc length, giving $\vec{i}=\dot{c}_v(s_u)$ and $\vec{j}=\dot{c}_u(s_v)$.

If the coordinate reference system is 2D, then the vector $\vec{n}$ extends the local coordinate system by supplying an "upward" height/elevation vector. In this case, the vector basis $\left(\vec{i},\vec{j}\right)$ must be a right hand system; that is to say, the counter-clockwise oriented angle from *i* to *j* must be less than 180°. This gives a right-handed "moving frame" for each curve of local coordinate axes given by $<\vec{i},\vec{j}>$. A moving frame is defined to be a continuous function from the geometric object to a basis for the local tangent space of that object. For the section curves, this is the derivative of the curve, the local tangent. For a parametric curve surface, this is a local pair of tangents.

NOTE The existence of a viable moving frame is the definition of "orientable" manifold. This is why the existence of a continuous upNormal implies that the surface is orientable. Non-orientable surfaces, such as the Möbius band and Klein bottle are counter intuitive. Subclause 6.4.25 forbids their use in application schemas conforming to this document. Klein bottles cannot even be constructed in 3D space, but require 4D space for non singular representations.

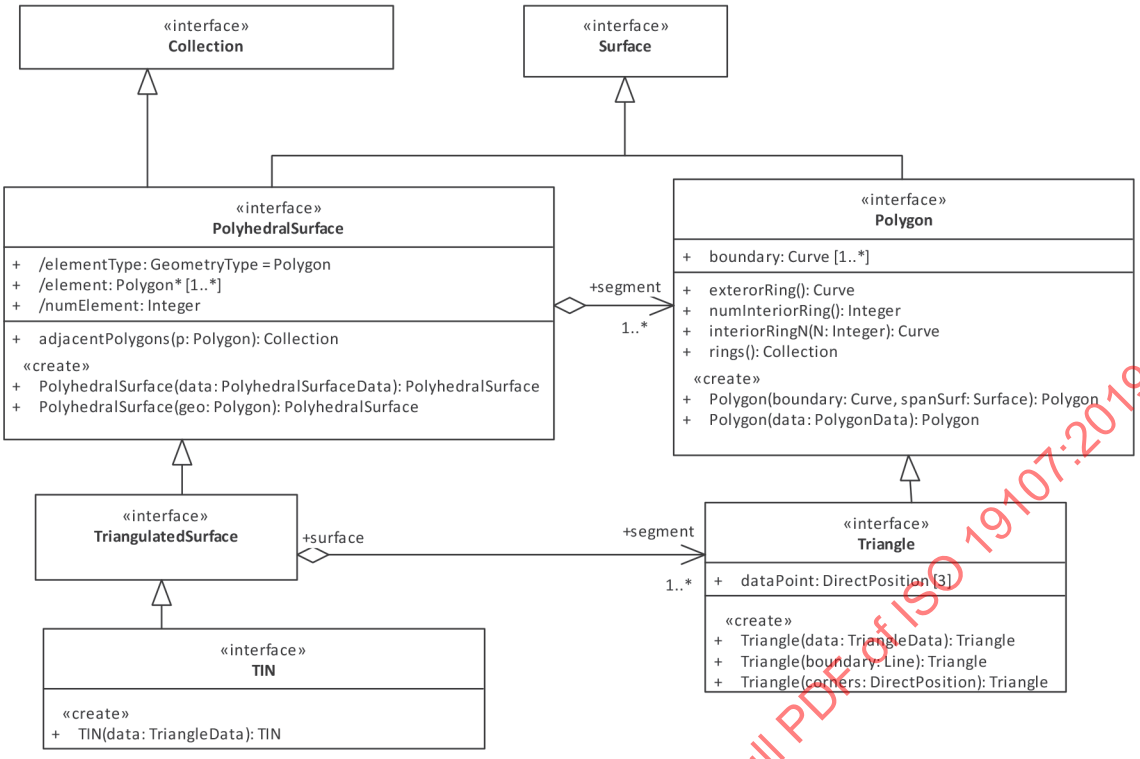



**Figure 28 — Polyhedral surface**

**Table 7 — Examples of parametric curve representations**

| Surface type | Horizontal curve type | Vertical curve type |
|---|---|---|
| Cylinder | Circle, constant radii | Line Segment |
| Cone | Circle, increasing radii from the apex point | Line Segment |
| Sphere | Circle of constant latitude | Circle of constant longitude |
| BilinearGrid | Line string | Line string |
| BicubicGrid | Cubic spline | Cubic spline |

The rows from the parameter grid are associated with dataPoints for the horizontal surface curves; the columns are associated with dataPoints for the vertical surface curves. The working assumption is that for a pair of parametric coordinates ($u$, $v$) that the horizontal curves for each integer offset are calculated and evaluated at "$u$". This defines a sequence of control points:

$$\{S(u,v_n)|\}=\{c_{v_n}(u):u=k_{0,n},\ldots k_{columns-1,n}\} \tag{105}$$

From this sequence, a vertical curve is calculated for "$u$", and evaluated at "$v$". The order of calculation (horizontal-vertical versus vertical-horizontal) does not normally make a difference. Where it does, the horizontal-vertical order should be used.

NOTE A common case of a parametric surface $S$ is a 2D spline. In this case, the weight functions for each parameter make order of calculation unimportant:

$$S(u,v)=\sum_{i=0}^{rows-1}\sum_{j=0}^{colums-1} w_i^u(u)w_j^v(v)\vec{P}_{i,j} \tag{106}$$

where $\vec{P}_{i,j}$ is the control point in the $i^{th}$ row and $j^{th}$ column.

Logically, any pair of curve interpolation types can lead to a subtype of ParametricCurveSurface.

#### 8.3.2.2 Attribute horizontalCurveType, verticalCurveType

The attribute "horizontalCurveType" indicates the type of section curves used to traverse the surface "horizontally" which fix the parameter "$v$" and uses the parameter "$u$", $S.horizontalCurve(v)(u) = C_v(u)$.

```
horizontalCurveType:GeometryType
```

**REQ. 187** **The GeometryType returned by horizontalCurveType shall be in the local codelist GeometryType and shall be a subtype of Curve.**

The attribute "verticalCurveType" indicates the type of surface curves used to traverse the surface vertically which fix the parameter "$u$" and uses the parameter "$v$", $SverticalCurve(u)(v) = C_u(v)$.

**REQ. 188** **The GeometryType returned by verticalCurveType shall be in the local codelist GeometryType and shall be a subtype of Curve.**

```
verticalCurveType:GeometryType
```

#### 8.3.2.3 Attribute rows

The attribute "rows" gives the number of rows in the parameter grid.

```
rows:Integer
```

#### 8.3.2.4 Attribute columns

The attribute "columns" gives the number of columns in the parameter grid.

```
columns:Integer
```

#### 8.3.2.5 Attribute dataPoints

The dataPoints are the functional images of the knots in the parameter grid.

```
dataPoints:DirectPosition[0..*]
```

$$\begin{aligned} &knots = (u_i, v_j), \\ &\text{dataPoint}(i,j) = S(u_i, v_j) = c_{u_i}(v_j) = c_{v_j}(u_i) \end{aligned} \tag{107}$$

#### 8.3.2.6 Attribute controlPoints

If either the horizontal or the vertical curves require control points in their definition, they may be accessed via the control point array. There will be (rows × columns) points in the control point array, stored in row-major form.

```
controlPoints:DirectPosition[0..*]
```

#### 8.3.2.7 Attribute horizontalCurveType

The curves associated to the points in a row of the controlPoint 2D array will normally be of a single curve type where the construction parameters vary across rows.

```
horizontalCurveType:GeometryType
```

#### 8.3.2.8 Attribute verticalCurveType

This attribute describes the "curves" that correspond to the columns of the controlPoint 2D array.

```
verticalCurveType:GeometryType
```

#### 8.3.2.9 Operation: horizontalCurve: Curve

The operation "horizontalCurve" constructs a curve that traverses the surface horizontally with respect to the parameter "*s*". This curve holds the parameter "*t*" constant.

```
horizontalCurve(v:Real):Curve
```

NOTE The Curve returned by this function or by the corresponding vertical curve function, are normally not part of any Geometry Complex to which this surface is included. These are, in general, calculated transient values. The exceptions to this can occur at the extremes of the parameter space. The boundaries of the parameter space support for the surface map normally to the boundaries of the target surfaces.

#### 8.3.2.10 Operation: verticalCurve: Curve

The operation "verticalCurve" constructs a curve that traverses the surface vertically with respect to the parameter "*t*". This curve holds the parameter "*s*" constant.

```
verticalCurve(u:Real):Curve
```

#### 8.3.2.11 Operation: surface: DirectPosition

The operation "surface" traverses the surface both vertically and horizontally.

```
surface(u:Real,v:Real):DirectPosition
```

### 8.3.3 Datatype ParametricCurveSurfaceData

ParametricCurveSurfaceData is the default constructor datatype for ParametricCurveSurface and contains datatypes for the five attributes of a ParametricCurveSurface defined in:

— 0 and 0 as “interpolation”,

— 0 as “rows”,

— 0 as “columns”,

— 0 as “dataPoint” and

— 0 as “controlPoint”.

### 8.3.4 Interface BilinearGrid

A BilinearGrid is a ParametricCurveSurface that uses line strings as the horizontal and vertical curves.

EXAMPLE On a 2 by 2 grid, the equations are quite symmetric (using integer knots, the data points at the knots are the data points for that particular crossing, (integer, integer) coordinate):

$$\begin{aligned} &\{S(0,0)=\vec{P}_{0,0}, S(1,0)=\vec{P}_{1,0}, S(0,1)=\vec{P}_{0,1}, S(1,1)=\vec{P}_{1,1}\}: \\ &S(u,v)=\vec{P}_{u,v}=(1-u)(1-v)\vec{P}_{0,0}+(1-u)v\vec{P}_{0,1}+u(1-v)\vec{P}_{1,0}+uv\vec{P}_{1,1} \end{aligned} \tag{108}$$

If the four points are coplanar, the section of the grid is a polygon.

NOTE This is not a polygonal surface, since each of the grid squares is a ruled surface, and not necessarily planar.

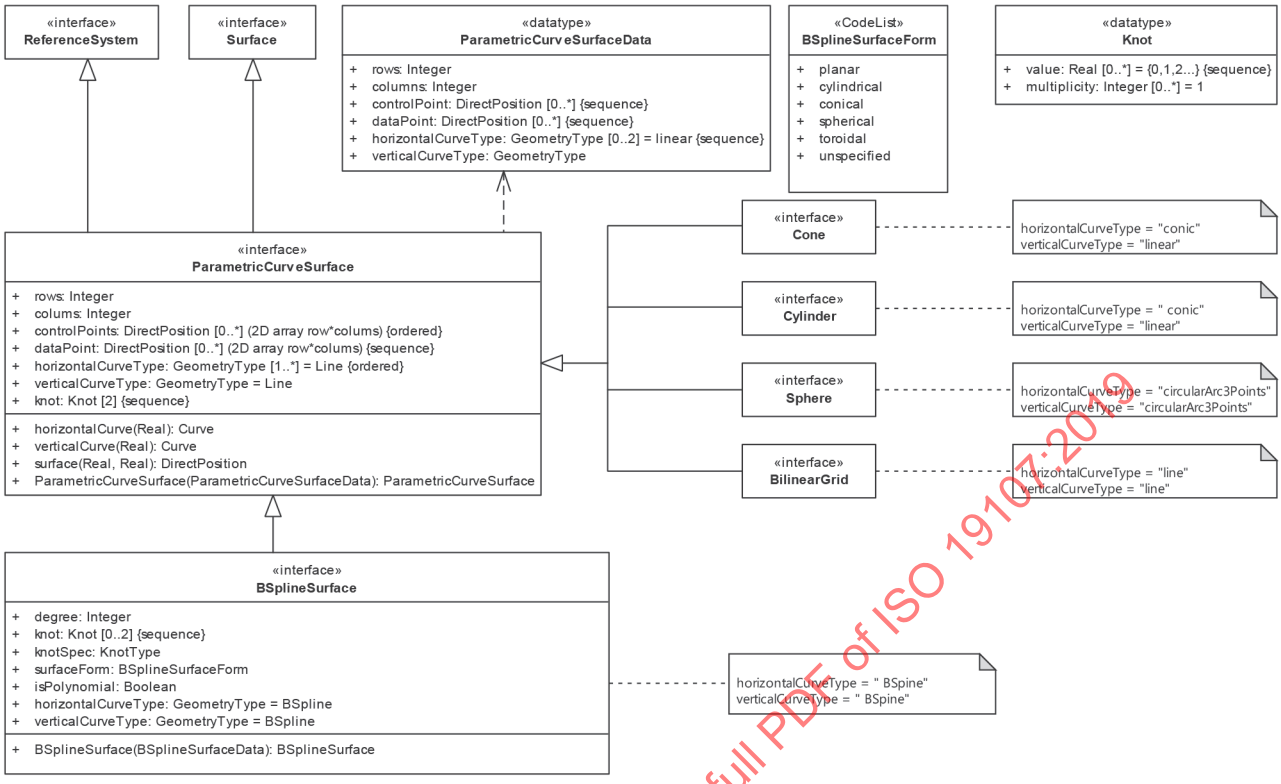



**Figure 29 — ParametricCurveSurface and its subtypes**

Another internal representation is the use of real valued polynomials (one for each coordinate dimension, as defined by the CRS.dimension @(0), on 2 variables (here we will use $(u, v) \in [a, b] \otimes [c, d]$)

### 8.3.5 Extensions of ParametricCurveSurface

More complex versions for interpolations of ParametricCurveSurface are available.

$$P(u,v)=\begin{bmatrix} p_0(u,v) \\ p_1(u,v) \\ \dots \\ p_{d-1}(u,v) \end{bmatrix}$$

$$p_i(u,v)=\begin{bmatrix} 1 & u & \dots & u^n \end{bmatrix}\begin{bmatrix} a^i_{00} & a^i_{01} & \dots & a^i_{0m} \\ a^i_{10} & a^i_{11} & \dots & a^i_{1m} \\ \dots & \dots & \dots & \dots \\ a^i_{n0} & & & a^i_{nm} \end{bmatrix}\begin{bmatrix} 1 \\ v \\ \dots \\ v^m \end{bmatrix}=\sum_{r=o,c=o}^{n,m} a^i_{rc}u^r v^c \quad (109)$$

## 8.4 Requirements Class Parametric Curve Surface Data

**REQ. 189** **An implementation of the requirements class Parametric Curve Data shall support all datatypes in Requirements Class Parametric Curve Surface**

## 8.5 Requirements Class Conic Surface

### 8.5.1 Semantics

A conic surface is either a cone or a surface derived from cutting a cone with a planar intersection.

**REQ. 190** **An implementation of the Requirements Class Conic Surface shall implement Requirements Class Geometry and at least one Curve Requirements Class.**

**REQ. 191** **An implementation of the Package Conic Surface shall have all instances and properties specified for this package, it contents, and its dependencies, contained in the UML model for this package in this document.**

### 8.5.2 Interface Sphere

A Sphere can be represented as a GriddedSurface given as a family of circles whose positions vary linearly along the axis of the sphere, and whose radius varies in proportion to the cosine function of the central angle. The horizontal circles resemble lines of constant latitude, and the vertical arcs resemble lines of constant longitude.

NOTE If the control points are sorted in terms of increasing longitude, and increasing latitude, the upNormal of a sphere is the outward normal.

EXAMPLE If we take a gridded set of latitudes and longitudes in degrees, (φ,λ), such as

| | | | | |
|---|---|---|---|---|
| (–90, –180) | (–90, –90) | (–90,0) | (–90,90) | (–90,180) |
| (–45, –180) | (–45, –90) | (–45,0) | (–45,90) | (–45,180) |
| (0, –180) | (0, –90) | (0,0) | (0,90) | (0, 180) |
| (45, –180) | (45, –90) | (45,0) | (45,90) | (45, 180) |
| (90, –180) | (90, –90) | (90,0) | (90,90) | (90, 180) |

And map these points to 3D using the usual formulae (where R is the radius of the required sphere).

$$x = \rho \cos\lambda \cos\varphi$$
$$y = \rho \cos\lambda \sin\varphi \quad (110)$$
$$z = \rho \sin\lambda$$

We have a sphere of radius $\rho$, centred at (0, 0), as a gridded surface. Notice that the entire first row and the entire last row of the control points map to a single point each in 3D Euclidean space, North and South poles respectively, and that each horizontal curve closes back on itself forming a geometric cycle. This gives us a metrically bounded (of finite size), topologically unbounded (not having a boundary, a cycle) surface.

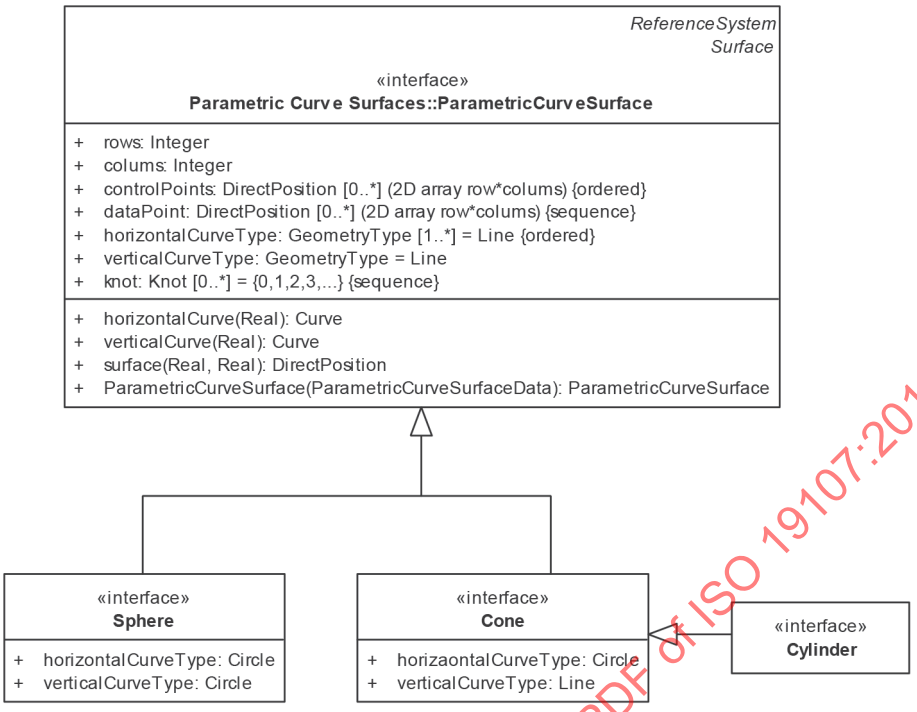



**Figure 30 — Conics**

**REQ. 192** **An implementation of the interfaces Cone and Cylinder shall have all properties specified for it in the UML model in this document.**

### 8.5.3 Interface Cone

A Cone can be represented as a GriddedSurface given as a family of conic sections whose dataPoints vary linearly.

NOTE A 5-point ellipse with all defining positions identical is a point. Thus, a truncated elliptical cone can be given as a 2 × 5 set of control points << *P*1, *P*1, *P*1, *P*1, *P*1 >. < *P*2, *P*3, *P*4, *P*5, *P*6 >>. P1 is the apex of the cone. P2, P3, P4, P5, and P6 are any five distinct points around the base ellipse of the cone. If the horizontal curves are circles as opposed to ellipses; then a circular cone can be constructed using << *P*1, *P*1, *P*1 >. < *P*2, *P*3, *P*4 >>.

### 8.5.4 Interface Cylinder

A Cylinder can be represented as a GriddedSurface given as a family of circles whose positions vary along a set of parallel lines, keeping the cross-sectional horizontal curves of a constant shape.

NOTE Given the same working assumptions as in the previous note, a Cylinder can be given by two circles, giving us control points of the form << *P*1, *P*2, *P*3 >. < *P*4, *P*5, *P*6 >>.

## 8.6 Requirements Class Conic Surface Data

**REQ. 193** **An implementation of the requirements class Conic Surface Data shall support all datatypes in Requirements Class Conic Surface.**

## 8.7 Requirements Class Spline Surface

### 8.7.1 Semantics

A spline surface is a type of gridded surface for which the spans of the gridded knot space is defined by spline functions.

**REQ. 194** **An implementation of the Requirements Class Spline Surface shall implement Requirements Class Geometry and Requirements Class Spline Curve.**

**REQ. 195** **An implementation of the Package Spline Surface shall have all instances and properties specified for this package, its contents and its dependencies, contained in the UML model for this package in this document.**

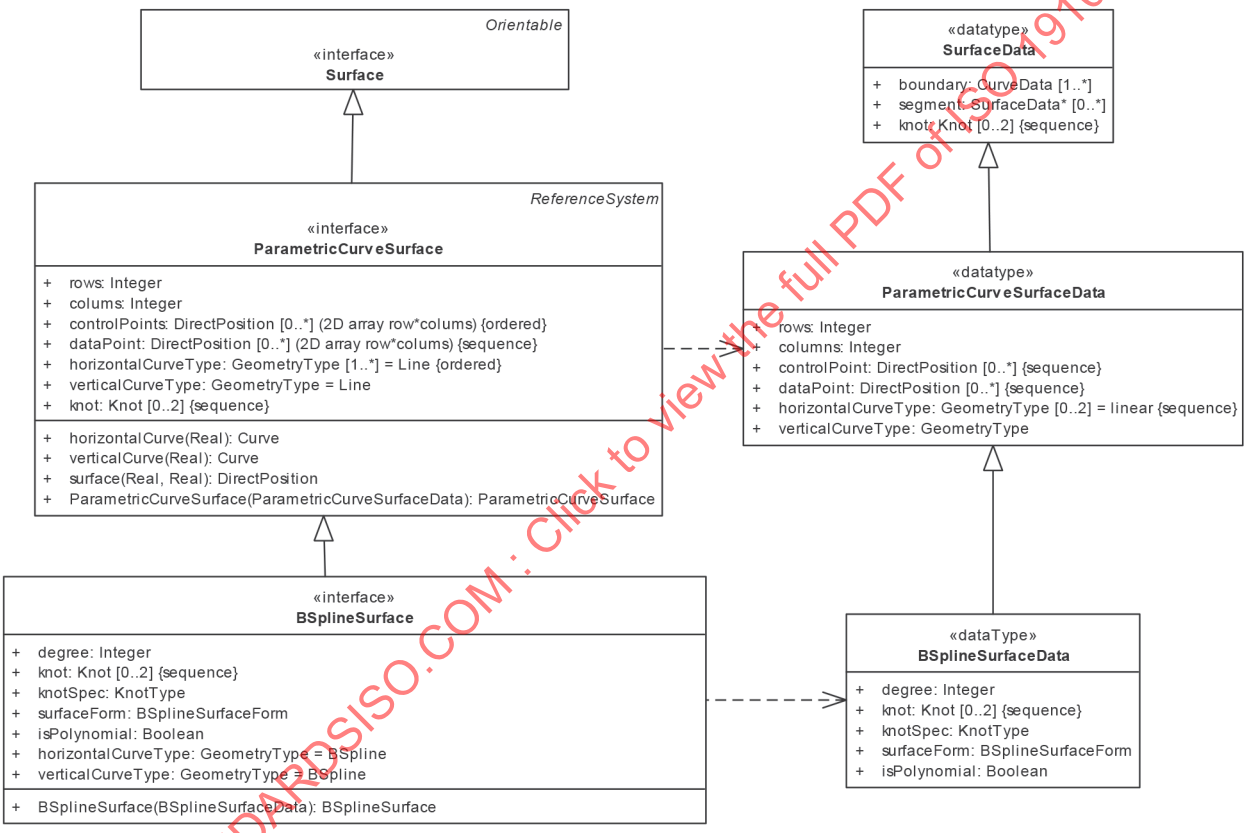



**Figure 31 — Parametric and BSpline Surfaces**



### 8.7.2 Interface BSplineSurface (and NURBS)

#### 8.7.2.1 Semantics

A B-spline surface (Figure 28, Figure 31) is a rational or polynomial parametric surface that is represented by control points, basis functions and possibly weights. If the weights are all equal, then the spline is piecewise polynomial. If they are not equal ("isPolynomial" is FALSE), then the spline is piecewise-rational and the CRS is augmented to be a homogeneous coordinate system with a "weight" column (see 7.13.4 above). If the Boolean "isPolynomial" is TRUE then CRS is not a homogeneous coordinate reference system.

The attributes "horizontalCurveType" and "verticalCurveType" inherited from ParametricCurveSurface, will be "BSpline." If the CRS of a b spline is homogeneous, the use of "BSpline" as a value for curve type is equivalent to "NURBS.

#### 8.7.2.2 Attribute degree: Integer

The attribute "degree" shall be the algebraic degree of the b-spline basis functions.

```
BSplineSurface::degree:Integer
```

#### 8.7.2.3 Attribute knot: Knot[2]

The attribute "knot" shall be two sequences of distinct knots used to define the b spline basis functions for the two parameters. Recall that the knot datatype holds information on knot multiplicity. There are two ways to display the knots, either as a simple array of real (in which case knots with multiplicities are repeated), or an array of pairs of (knot value, knot multiplicity).

```
BSplineSurface::knot:Knot[2]
```

**Per. 4 Implementations may choose a single knot representation**

#### 8.7.2.4 Attribute surfaceForm: BSPlineSurfaceForm

The attribute "surfaceForm" is used to identify particular types of surface that this spline is being used to approximate. It is for information only, used to capture the original intention. If no such approximation is intended, then the value of this attribute is "unspecified."

```
BSplineSurface::surfaceForm:BSplineSurfaceForm
```

#### 8.7.2.5 Attribute knotSpec: KnotType

The attribute "knotSpec" gives the type of knot distribution used in defining this spline. This is for information only and is set according to the different construction-functions. See 7.13.2.

```
BSplineSurface::knotSpec:KnotType
```

#### 8.7.2.6 Attribute isPolynomial: Boolean

The attribute "isPolynomial" is "True" if this is a polynomial spline. If "False", the spline is rational, i.e. a NURBS "non-uniform rational b-spline". Regardless of the name, a NURBS curve can be uniform. Rational b-splines use homogeneous coordinates.

```
BSplineSurface::isPolynomial:Boolean
```

#### 8.7.2.7 BSplineSurfaceData (default constructor datatype)

The class constructor "BSplineSurface" takes the pertinent information described in the attributes above and constructs a B spline surface. If the knotSpec is not present, then the knotType is uniform and the knots are evenly spaced, and, except for the first and last, have multiplicity = 1. At the ends the knots are of multiplicity = degree+1. If the knotType is uniform, they need not be specified.

```
degree:                 Integer
knot:                   Knot[2]                          // 2 knot arrays
knotSpec:               KnotType
surfaceForm:            BSplineSurfaceForm[0,1]
isPolynomial:           Boolean
```

### 8.7.3 Codelist BSplineSurfaceForm

The codelist "BSplineSurfaceForm" shall be used to indicate a particular geometric form represented by a BSplineSurface. The potential values are:

— planar — a bounded portion of a plane represented by a B-spline surface of degree 1 in each parameter.

— cylindrical — a bounded portion of a cylindrical surface represented by a B-spline surface.

— conical — a bounded portion of the surface of a right circular cone represented by a B-spline surface.

— spherical — a bounded portion of a sphere, or a complete sphere represented by a B-spline surface.

— toroidal — a torus or a portion of a torus represented by a B-spline surface.

— unspecified — no particular surface is specified.

```
BSplineSurfaceForm::
                                    planar
                                    cylindrical
                                    conical
                                    spherical
                                    toroidal
                                    unspecified
```

## 8.8 Requirements Class Spline Surface Data

**REQ. 196** **An implementation of the requirements class Spline Surface Data shall support all datatypes in Requirements Class Spline Surface.**



# 9 Interpolations for Solids

## 9.1 Requirements Class Boundary Representation Solid

The simplest representation of a solid in a 3D space is by specification of its boundary surfaces. By the Jordan separation theorem, a closed simple 3D surface divides a $\mathbb{E}^3$ space into 2 solid components, one finite and one infinite. In other coordinate systems, the normals to the surface point outward from the defined solid So there is no ambiguity in terms of which solid is being defined.

The root interface Surface (Figure 32) is sufficient to support boundary representations, where a solid, like a polygon, is represented by its boundary. This interface can be support by an application only if at least one surface requirements class with realizable surfaces is also supported.

**REQ. 197** **An implementation of the Requirements Class Boundary Representation Solid shall implement Requirements Class Geometry and at least one Surface Requirements Class.**

**REQ. 198** **An implementation of the Boundary Representation shall have all instances and properties specified for this package, its contents, and its dependencies, contained in the UML model for this package in this document.**

The interface Solid defined in 6.4.28 is sufficient to support a boundary representation.

**REQ. 199** **An implementation of the interface Solid shall have all properties specified for it in the UML model in this document.**

## 9.2 Requirements Class Boundary Representation Solid Data

**REQ. 200** **An implementation of the requirements class Boundary Representation Solid Data shall support all datatypes in Requirements Class Boundary Representation.**

## 9.3 Requirements Class Parametric Curve Solid

### 9.3.1 Interface ParametricCurveSolid

#### 9.3.1.1 Semantics

This interface does for solids what the interface ParametricCurveSurface (8.3.2) does for surfaces, it builds solids by bundling curves.

For any valid interpolation, this creates a “rectangular” parameter space of the same dimension of the geometry that mapped, for all parametric curves, surfaces or solids. The “knot” coordinate systems to the geometry being defined. This will allow a mechanism to map information associated to the knot coordinate to the corresponding positions in the coordinate reference system in a manner similar to or extending those in ISO 19148:2012 (see [8]).

#### 9.3.1.2 Attribute: horizontalCurveType, verticalCurveType, depthCurveType

**REQ. 201** **An implementation of the Requirements Class Parametric Curve Solid shall implement Requirements Class Geometry and at least one Curve Requirements Class.**

**REQ. 202** **The GeometryType returned by horizontalCurveType, verticalCurveType and depthCurveType shall be in the local codelist GeometryType and shall be a subtype of Curve.**

The attribute "rows" gives the number of horizontal rows in the parameter grid. The attribute "columns" gives the number of vertical columns in the parameter grid. The attribute "files" gives the number of depth files in the parameter grid

```
ParametricCurveSolid::rows:Integer

ParametricCurveSolid::columns:Integer

ParametricCurveSolid::files:Integer
```

### 9.3.1.3 Attribute: dataPoint, controlPoint

The dataPoints are the functional images of the knots in the 3D parameter grid.

$$knots\left(u_i, v_j, t_k\right),$$
$$dataPoint\left(i,j,k\right)=S\left(u_i, v_j, t_k\right)=c_{u_i,t_k}\left(v_j\right)=c_{u_i,v_j}\left(t_k\right)=c_{v_j,t_k}\left(u_i\right) \tag{111}$$

If either the horizontal, vertical or the depth curves require control points in their definition, they may be accessed via the control point array.

```
dataPoint:  DirectPosition[rows × columns × files]

controlPoint: DirectPosition[rows × columns × files]
```

There will be (rows × columns × files) points in the control point array, stored in row-major form.

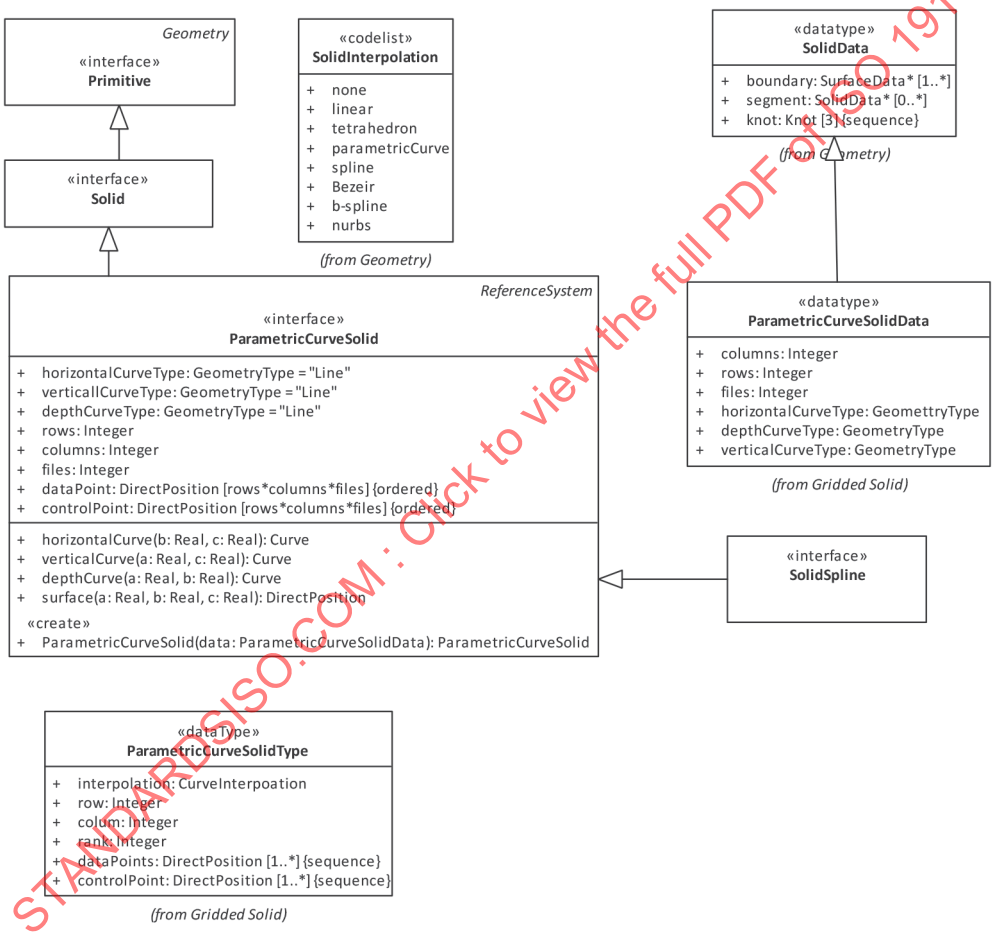



**Figure 32 — Solid**

### 9.3.2 Interface BSolidSpline

Solid splines are 3D parametric curve solids, where the various curves are splines (b-splines normally), and the equation for the solid interior is:

$$s_{p,q,r}\left(u,v,t\right)=\sum_{i=0}^{p}\sum_{j=0}^{q}\sum_{k=0}^{r}N_{i,p}\left(u\right)N_{j.q}\left(v\right)N_{k,r}\left(t\right)\overrightarrow{P_{i,j,k}} \tag{112}$$

### 9.3.3 Other interpolations

Tetrahedron interpolation creates a Delaunay tetrahedral network in the (*u, v, t*) where each tetrahedron is a subset of a cube in the (u, v, t) grid. The interpolation uses barycentric interpolation to map the parameter space to the coordinate space.

## 9.4 Requirements Class Parametric Curve Solid Data

**REQ. 203** **An implementation of the requirements class Parametric Curve Solid Data shall support all datatypes in Requirements Class Parametric Curve Solid.**

# 10 Topology

## 10.1 Requirements Class Topology root

### 10.1.1 Semantics

The most productive use of topology is to accelerate computational geometry. The method by which this is accomplished is to associate explicitly feature instances and geometric object instances in a manner consistent with and derived from their implicit geometric relations. In some cases, these associations are derived from a conceptual geometry that does not agree with the representation of the feature instances. For this purpose, it is necessary to define topology packages that parallel the geometry packages in 5. Figure 33 shows these packages and their dependencies.

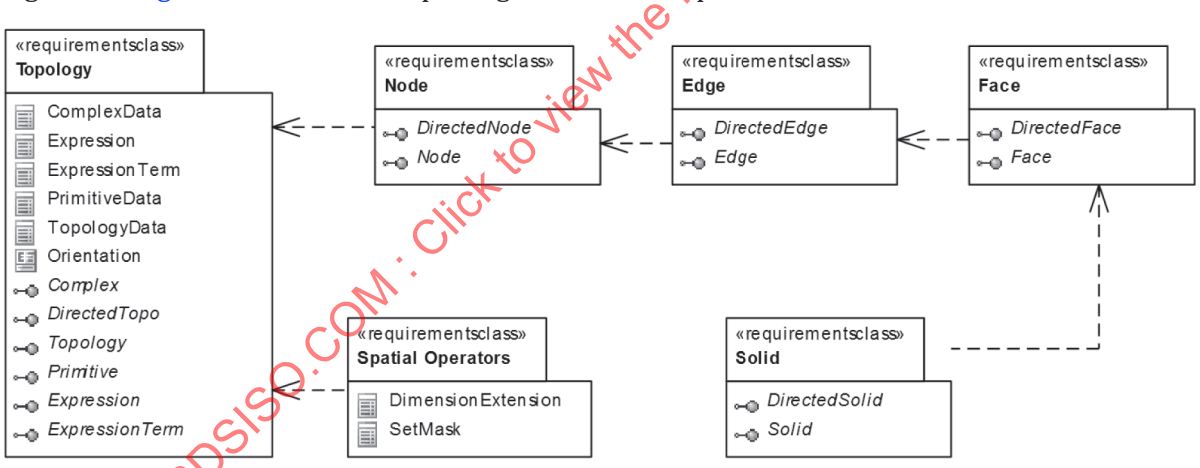



**Figure 33 — Topology packages and internal dependencies**

Figure 34 gives an overview of the class structure of the basic topological packages. The root class of the diagram is Topology::Object. Under this, they are Primitive, and Complex, which are related in way similar to the Geometry Primitive and Geometry Complex, so that a Complex is an organized structure of Primitives. The major difference being that a Geometry Primitive is more loosely coupled to a Geometry Complex, allowing it to stand alone, whereas a Primitive must be in at least one Complex. An instance of DirectedTopo shall contain a reference to a Primitive and an orientation parameter, similar to the Orientable in 6.3.13. Since only two orientations are possible, regardless of dimension, each primitive is associated with two directed topological. To conserve on the number of objects and to make the natural identification of a primitive with its positive orientation, each primitive in each dimension is subclassed under its corresponding directed topological object. A common minimalist topological structure "MiniTopo" is described in Annex C.

— Geometric calculations such as containment (point in polygon), adjacency, boundary and network tracking are computationally intensive. For this reason, combinatorial structures known as topological complexes are constructed to convert computational geometry algorithms into

combinatorial algorithms. Another purpose is, within the geographic information domain, to relate feature instances independently of their geometry. For the first purpose, topology definitions in this clause parallel the structure of the geometric definitions in Geometry (Clause 5 through 8.8). For the second purpose, the classes in these packages are specified so that they can be used independently of the geometry.

— A topological complex consists of collections of topological primitives of all kinds up to the dimension of the complex. Thus, a 2-dimensional complex must contain faces, edges, and nodes, while a 1-dimensional complex or graph contains only edges and nodes.

NOTE Topological primitives are equivalent to but are not subclasses of geometric primitives. This is consistent with the view that topological complexes are constructed to optimize computational geometry procedures by the use of combinatorial algorithms. This also permits the creation of structures that ignore geometric constraints by using a topological complex that is not realized by a geometric complex.

The key to understanding the use of computational topology is to see the related procedures in both systems. As Figure 34 shows, there is a great deal of parallelism between how primitives and complexes are related in the two class systems.

The topological system is based on algebraic manipulations of multivariate polynomials. The definitions of the procedures, functions and operations in the topology packages are done so that geometric problems in the geometric domain can be translated into algebraic problems in the topology domain, solved there, and the solutions translated back to the geometric domain. A topological expression in this algebra is a multivariate, degree one polynomial, where the variables correspond to topological primitives.

The diagram in Figure 35 summarizes the relation between topology and geometry. The OCL constraint means that the diagram commutes such that navigation between Topology and Geometry preserve set theory and the topological properties of the Geometry. This mapping between Geometry and Topology allows problems solved in either domain leads to the same solutions.

**REQ. 204** **An implementation of the Requirements Class Topology root shall implement Requirements Class Geometry.**

**REQ. 205** **An implementation of the Package Topology Root shall have all instances and properties specified for this package, its contents and its dependencies, contained in the UML model for this package in this document.**

### 10.1.2 Interface Topology

#### 10.1.2.1 Semantics

Topology is an abstract class that supplies a root type for topological complexes and topological primitives.

Logically and structurally, topological objects and geometric objects could share the same subclass structure, but since there is a categorical homomorphism from topology to geometry that preserves boundary operations, this approach could cause confusion between the boundary of a topological object and the boundary of the corresponding geometric object. While the two mechanisms share many computational characteristics, as demonstrated by the homomorphism, they are different operations and need to be clearly separated.

#### 10.1.2.2 Attribute isEmpty

The attribute isEmpty shall behave in the same manner as the Geometry.IsEmpty (0).



**Rec. 12** **Implementation should use the isEmpty Boolean to set objects to NULL or Empty temporarily until a "transaction" or similar action is either completed or undone. Long-term existence (persistence) of an "empty" topology is not recommended but is not prohibited.**

**REQ. 206** **Any topology object for which isEmpty=TRUE, shall behave appropriately in its other operations and attributes.**

**REQ. 207** **The topological dimension of an empty topology shall be –1.**

**REQ. 208** **Empty topology objects shall only be associated to empty geometry objects.**

#### 10.1.2.3 Attribute dimension

**REQ. 209** **The integer returned by the attribute "dimension" shall be the topological dimension of this Topology. It shall be solely dependent on the instantiated class of the object and shall not be changed for a particular object without changing that object's class.**

For example, the value for dimension is 0 for nodes, one for edges, two for faces, and three for solids. Any object associated with this Topology shall have this same dimension.

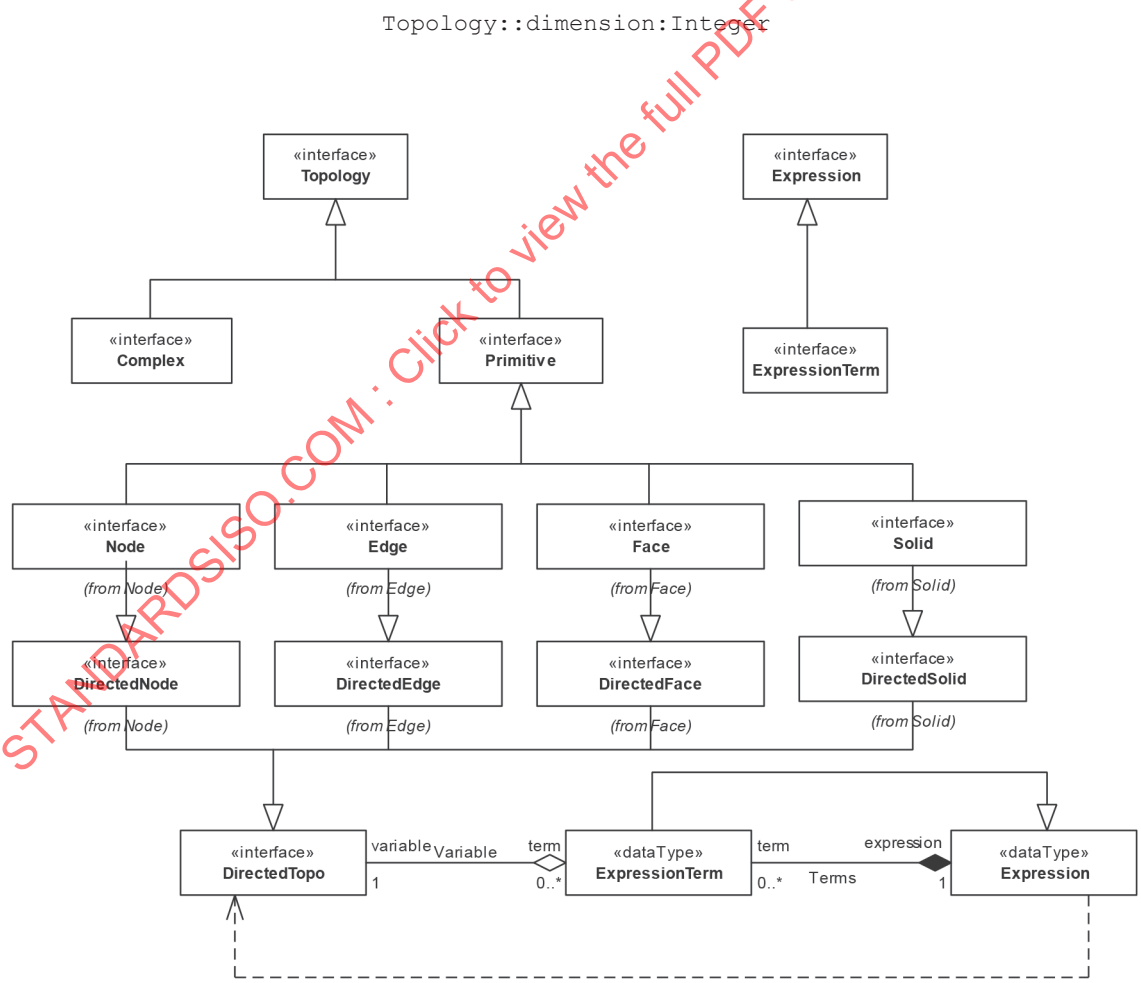



**Figure 34 — Topological class diagram**

#### 10.1.2.4 Attribute boundary

**REQ. 210** **The attribute "boundary" shall be a set of DirectedTopo structured as a Boundary that represents the boundary of the Topology.**

```
Topology::boundary:Boundary
```

**REQ. 211** **If this Topology is associated with an object, its boundary shall be consistent in orientation with that object as described in the geometry packages.**

As a constraint, the dimension of a boundary shall always be one less than the dimension of the original object. For this reason, the dimension of the empty set shall be considered to be "–1".

```
Topology:

    boundary.dimension=this.dimension()- 1
```

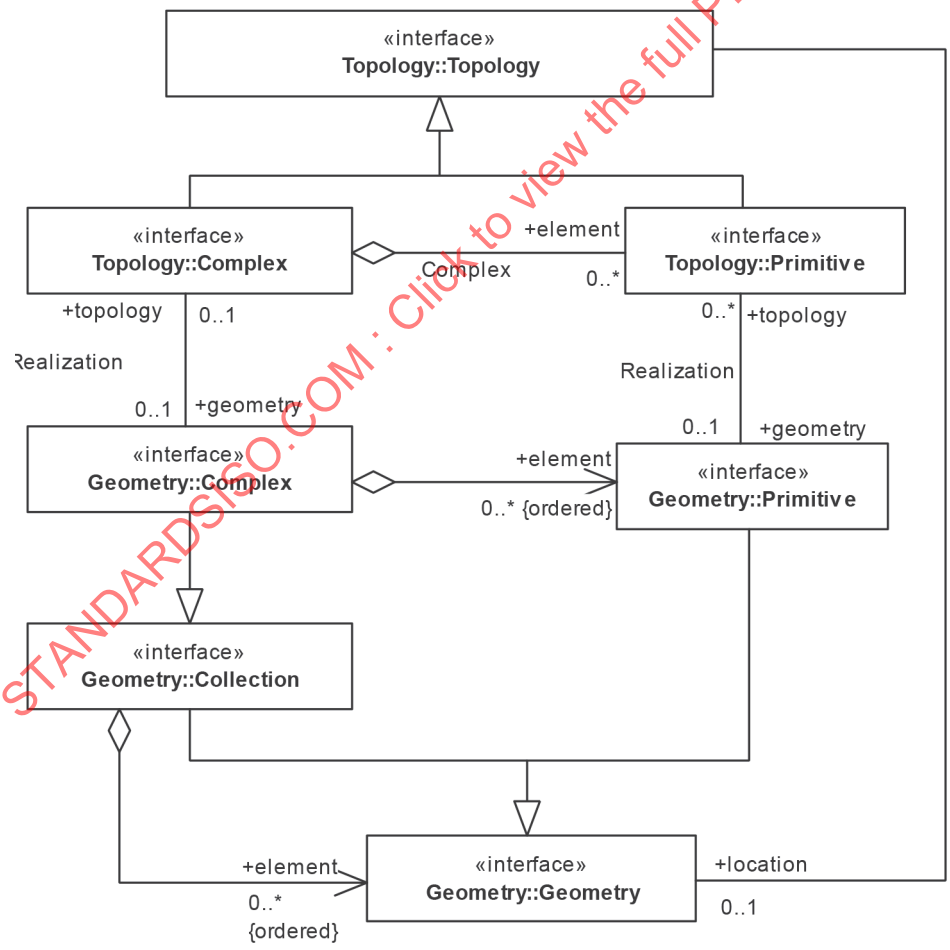



**Figure 35 — Relation between Geometry and Topology**

Figure 37 shows how the boundary function can be visualized as an association from objects of each dimension to objects of one less dimension.

In most cases, the return value will be a valid value of an Expression. The boundary returned can fail to be a valid Expression because of the requirement for simplest terms. A dangling or isolated edge in a face (one that has the same face on both sides) would cancel out in the conversion to a topological expression.

#### 10.1.2.5 Attribute coBoundary

**REQ. 212** **The attribute "coBoundary" shall have a value of a Set of DirectedTopo that represents all the Topology::Topologies that have this Topology on their boundary.**

In most cases, the return value will be a valid value of an Expression (10.1.5). An exception to this is when the corresponding object is on the boundary of a closed object (such as a curve that begins and ends at the same point). The Topology corresponding to that object would appear in the Set of DirectedTopo twice with opposite orientations and therefore cancel out when the coBoundary is cast from Set of DirectedTopo to Expression.

```
Topology::coBoundary: DirectedTopo[0..*]
```

Figure 37 illustrates how this attribute can be visualized as a relation between dimension levels of the Primitives, similar to the boundary operation, but directed in the opposite direction, increasing dimension instead of reducing it.

#### 10.1.2.6 Attribute interior

**REQ. 213** **The attribute "interior" shall have a value of the finite set of Primitives that comprises the interior of this object within the maximal complex of this object.**

For a Primitive this will be a self reference. For a Complex this will be all primitive elements in the Complex not on the boundary of the Complex. This is the homomorphic equivalent of the interior of a geometric realization of this Topology.

```
Topology::interior:Primitive[0..*]
```

#### 10.1.2.7 Attribute exterior

The attribute exterior" shall return the finite set of Primitives that comprises the exterior of this object within the maximal complex of this object. This consists of all Primitives in the maximal Complex that are not in the interior or the boundary of this Topology

```
Topology::exterior(): Primitive
```

#### 10.1.2.8 Attribute closure

The attribute "closure" is often useful; it is defined as a union of the interior and boundary of an object, and is thus not required in a basic implementation.

```
Topology::closure=interior.union(boundary)
```

#### 10.1.2.9 Attribute maximalComplex

**REQ. 214** **The attribute "maximalComplex ()" shall have a value equal to the maximal (largest) Complex that contains this Topology.**

```
Topology::maximalComplex:Complex
```

**REQ. 215 A Topology shall be included in one and only one maximal Complex.**

NOTE A complex is maximal if it is contained in no larger complex. The cardinality restriction implied by this operation means that any Topology is in one and only one maximal complex.

### 10.1.3 Interface Primitive

#### 10.1.3.1 Semantics

Topological primitives (Figure 38) are the non-decomposed elements of a topological complex. As such, they normally correspond to the geometric primitives of a like dimension that are the components of a geometric complex. When a geometric complex is the realization of a topological complex, then the primitives in each shall be in a dimension-preserving, one-to-one correspondence.

#### 10.1.3.2 Association: Realization

The association "Realization" links this Topological::Primitive to the Geometry::Primitive that it represents in its maximal complex. If this Topological::Primitive is used to describe a logical topological structure that is not realized by a Geometry Complex, then this relationship shall be empty for all Primitives contained in this Primitive's maximal Complex. Each Geometry Primitive may be associated with at most one Primitive in any Complex. If this Primitive is in any realized Complex, then it shall be associated with exactly one Geometry Primitive. A Geometry Primitive may be associated with different Primitives in different Complexes.

```
Primitive::geometry:Geometry::Primitive[0..1]

Geometry::Primitive::topology:Topological::Primitive[0..*]
```

#### 10.1.3.3 Association: Complex

The association "Complex" shall link this Primitive to the finite set of Complexes that contain it. Every Primitive shall be in some number of Complexes that are all subcomplexes of a unique maximal Complex containing this Primitive.

```
Primitive::complex:Complex[0..*]

Complex::element:Primitive[0..*]
```

#### 10.1.3.4 Association: Isolated In

All of the adjacency relations in topology between primitives whose dimensions differ by one or 0 are handled by the boundary and coboundary attributes. These attributes deal with instances of one primitive lying on the boundary of another primitive of one higher dimension, or with instances of the same dimension that share a common boundary element.

This includes instances where a "dangling" edge has the same face on both sides, or a "dangling" face has the same solid on both sides. The exception to this is when one primitive is surrounded by a primitive of at least two higher dimensions, with no intermediate primitive. These are truly isolated. In faces, this includes nodes that are not attached to an intermediate edge on the boundary of that face. In a 3D space, the isolated node could be connected to another edge that is not on the boundary of the surface in question, such as in the case where the edge is realized by a curve perpendicular to the surface that the face realizes. In solids, this can include nodes or edges that are not attached to surfaces in the boundary of the solid.



```
Primitive::isolated:Primitive[0..*]

Primitive::container:Primitive[0..1]

Primitive:

isolated.dimension < self.dimension- 1;

container.count=0 implies Primitive→exists(boundary.
topo→includes(self))
```

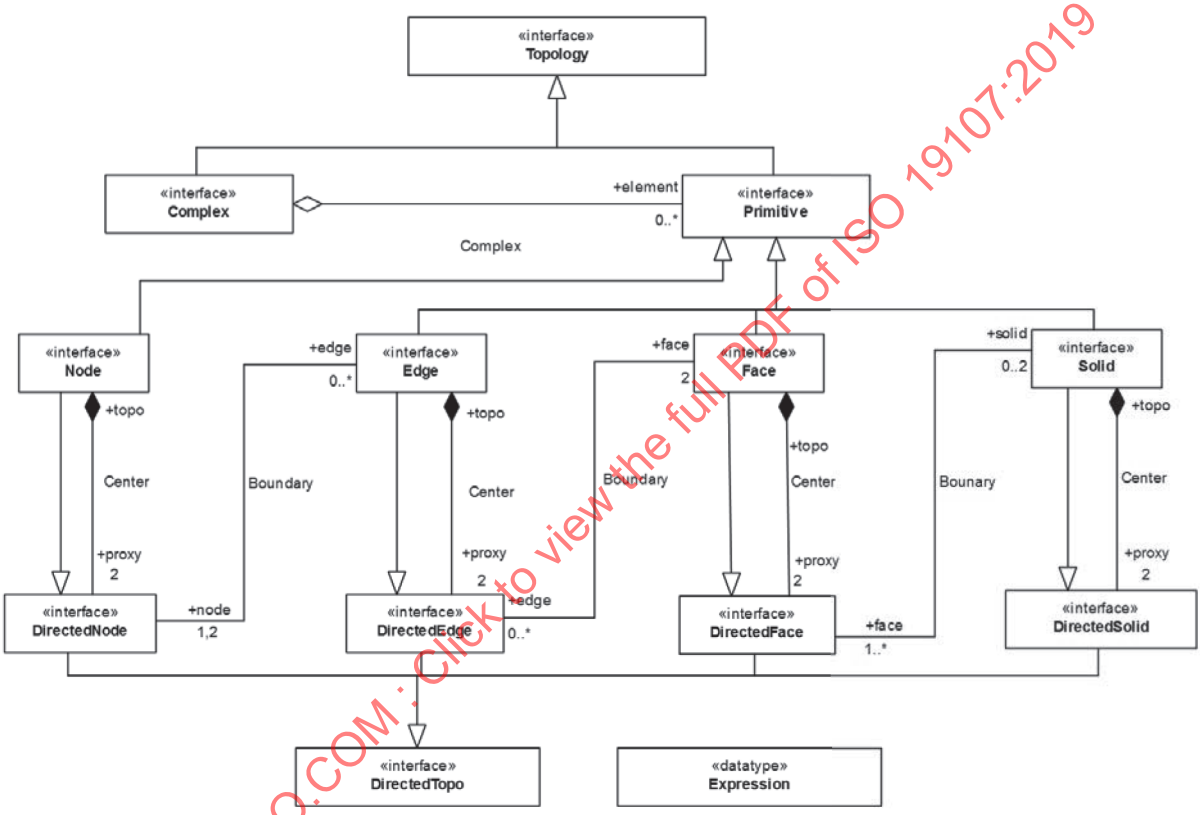



**Figure 36 — Topology and subclasses**

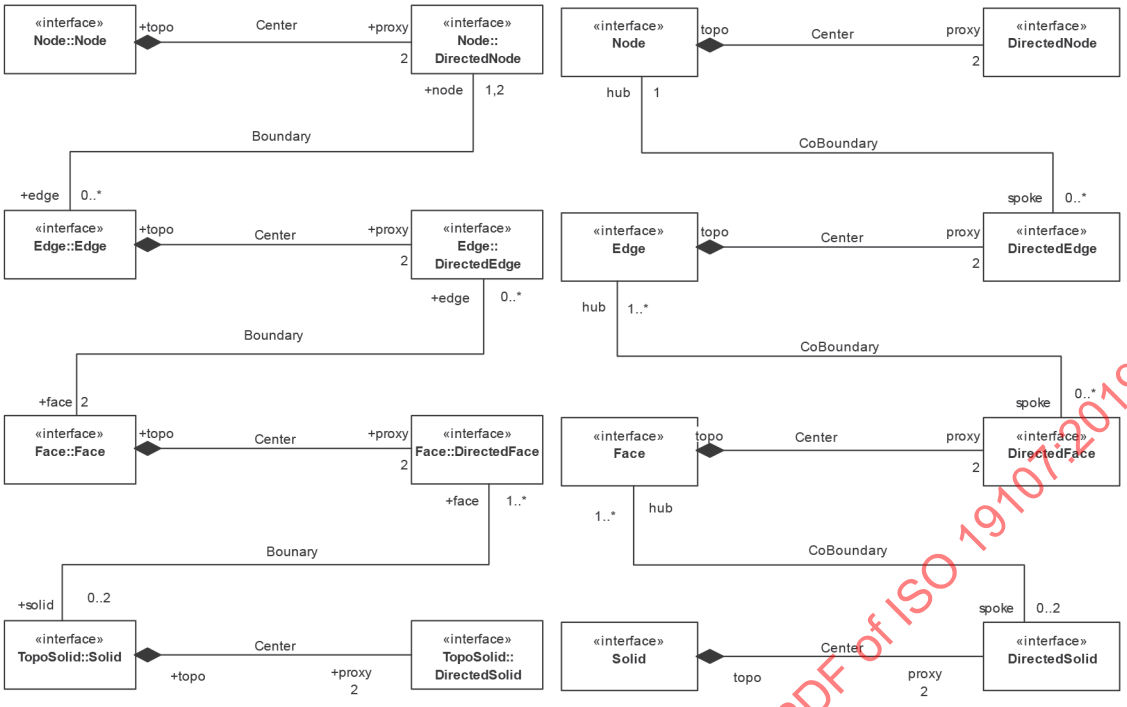



**Figure 37 — Boundary and coboundary operation represented as associations**

### 10.1.4 Interface DirectedTopo

#### 10.1.4.1 Semantics

From a computational point of view, elements of DirectedTopo (Figures 34, 36, 37) are equivalent to the various orientable geometric objects (Orientable) in the geometry packages (Curve and Surface). DirectedNode and DirectedSolid do not have separate geometric object equivalents.

As in the geometry, each topological primitive inherits from its corresponding directed topological primitive, but it satisfies more constraints. This means that Node is equivalent to a positive DirectedNode, an Edge to a positive DirectedEdge, etc.

NOTE An alternative type hierarchy would have separated Primitive and DirectedTopo, which would have entailed three objects for each primitive: the primitive itself, its equivalent positive directed topological primitive, and its reversal (a negative directed) topological primitive. This alternative is a valid implementation of the abstract types in this model, but it does not emphasize the logical equivalence of a topological primitive and its positive directed topological primitive. From an algebraic point of view, the subclassing and OCL constraints that identify a primitive with its positive directed primitive make it equivalent to the standard interpretation of the unary "+" (plus) in algebra as in "x = + x". Since the most powerful use of topological objects is in their symbolic manipulation, maintaining an algebraic metaphor is appropriate.

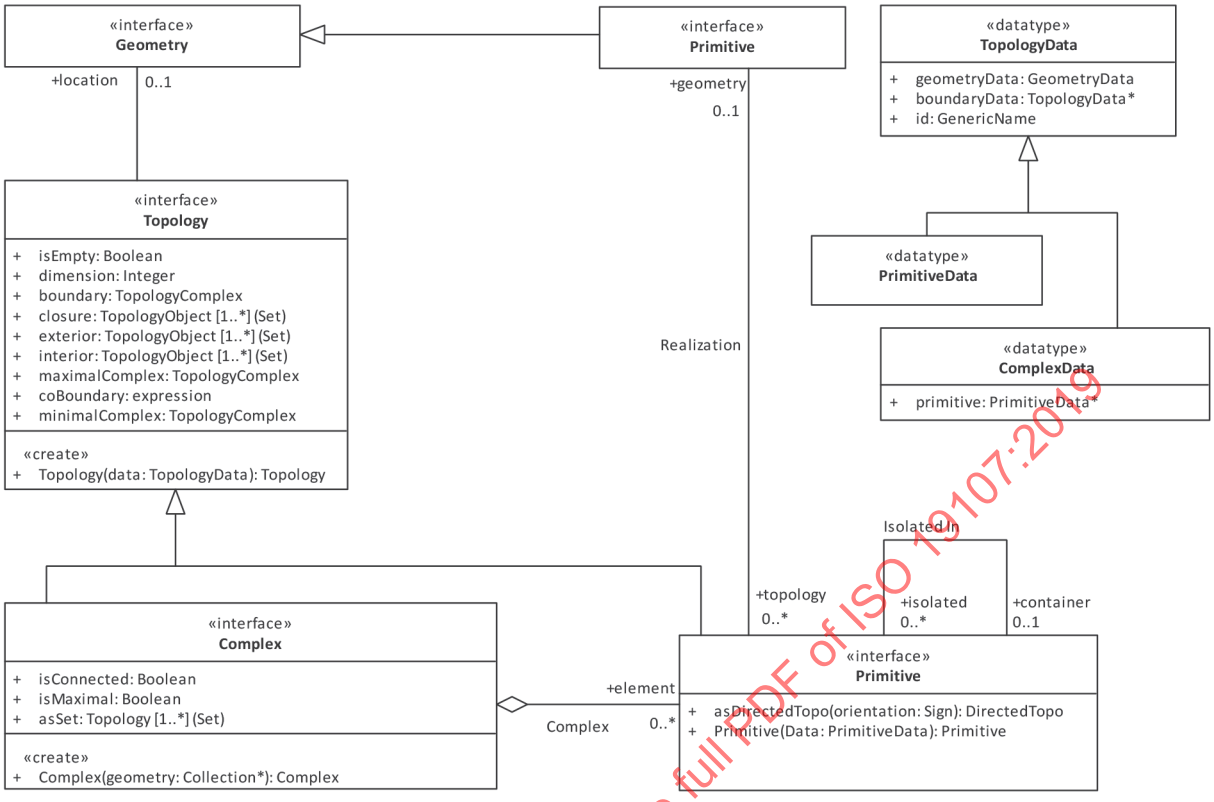



**Figure 38 — Topology, Complex and Primitive**

There is an implicit relation between the directed topological objects of adjacent dimensions. The boundary and coboundary use them to carry the same orientation sense. Thus if a positive directed edge is on the boundary of a face, then the positive **directed face** is on the coboundary of the associated edge. If a positive directed node is on the boundary of an edge, then the corresponding positive directed edge is on the coboundary of the associated node.

#### 10.1.4.2 Attribute orientation

**REQ. 216** **The attribute "orientation" shall be the sense in which this directed topological object is related to its underlying Primitive.**

```
DirectedTopo::orientation:Sign="+"
```

#### 10.1.4.3 Attribute negate

**REQ. 217** **The attribute "negate" shall be the opposite orientation of this primitive.**

```
DirectedTopo::negate:DirectedTopo
```

#### 10.1.4.4 Attribute expression

**REQ. 218** **The attribute "asExpression" shall be an Expression from this DirectedTopo, and shall retain the sign and the sense of the orientation. The derivation of the attribute shall be equivalent of the constructor from the interface Expression.**

```
DirectedTopo::expression:Expression
```

#### 10.1.4.5 Association: Centre

**REQ. 219** **The role "topo" in the association "Centre" shall identify the associated Primitive.**

**REQ. 220** **The inverse role "proxy" shall identify the two DirectedTopo instances associated with the particular Primitive.**

```
DirectedTopo::topo:Topology::Primitive

Primitive::proxy:DirectedTopo[2]
```

#### 10.1.4.6 Constraints

**REQ. 221** **Following the logic of the semantics of directed topological objects, the associated Topology for each directed topological object shall be of the appropriate type.**

```
DirectedNode: topo.isKindOf(Node);

DirectedEdge: topo.isKindOf(Edge);

DirectedFace: topo.isKindOf(Face);

DirectedSolid: topo.isKindOf(Solid);
```

NOTE These constraints use the OCL operator "isKindOf" to indicate that the class of a directed topological primitive corresponding to a topological primitive must be a realization of the corresponding topological primitive type.

The Centre association forms an important part of the algebra of the boundary and coBoundary operations.

```
DirectedTopo [boundary=(orientation)*topo.boundary]

Primitive [boundary=(proxy.orientation)*proxy.boundary]

DirectedTopo    negate.topo=topo;

   negate.orientation < > orientation;
```

### 10.1.5 Datatype TopologyData

#### 10.1.5.1 Semantics

The datatype “TopologyData” is a generic datatype to use in constructing topology after the underlying geometry is such that all geometry interiors and disjoint and all geometry boundaries can be represented by other geometries.

#### 10.1.5.2 Attribute id: GenericName

The attribute “id” is a referenceable identifier to use to create pointers in other datatypes to this instance.



```
id: GenericName
```

**REQ. 222** **The attribute "TopologyData: id" shall have a namespace component that is unique to its maximal complex and to its topological dimension.**

#### 10.1.5.3 Attribute boundaryData

The attribute "boundaryData" is an array of the referenceable identities of other TopologyData instances representing the geometries of items on the boundary of this element.

```
boundaryData: TopologyData []
```

#### 10.1.5.4 Attribute geometryData

The attribute "GeometryData" is a GeometryData representing the interior of this element. The dimension of the geometry will usually be consistent with the dimension of the topological dimensions, but not necessarily so in cases where the topology is being used to describe connections or similar artefacts such as in a Poincaré duality graph.

```
geometryData: GeometryData
```

### 10.1.6 DataType PrimitiveData

The datatype "PrimitiveData" for topology is a generic datatype to use in constructing primitive topology elements after the underlying geometry is such that all primitive geometry interiors and disjoint and all geometry primitive boundaries can be represented by other geometries. It inherits all needed information from the attributes of its supertype TopologyData.

### 10.1.7 DataType ComplexData

#### 10.1.7.1 Semantics

The datatype "ComplexData" for topology is a generic datatype to use in constructing complexes from a set of related primitive topology elements after the underlying geometry complex is such that all geometry primitives interiors and disjoint and all geometry boundaries can be represented by other geometries. It inherits most needed information from the attributes of its supertype TopologyData except a list of its primitives.

#### 10.1.7.2 Attribute primitive

The attribute "primitive" is a list of references to PrimitiveData datatype elements representing the primitive parts of this complex.

```
primitive: PrimitiveData []
```

### 10.1.8 Datatype Expression

#### 10.1.8.1 Semantics

Algebraic or computational topology is most easily conceptualized as the manipulation of multivariate, degree-one polynomials where the variables correspond to Primitives. The DirectedTopo class represents the terms in this algebra. The Expression class (Figure 39) represents the polynomial expressions.

The order of the terms in a polynomial does not affect its value, so the Expression class has been subclassed from a set of pairs (number, DirectedTopo). The operations of the Expression class are those needed to construct, manipulate and test these "polynomials".

The key to computational topology is the ability to treat pieces of topology in an algebraic or combinatorial manner. The primitives in this algebra are the Primitives. The monomials (single variable, single term polynomials) are the instances of Primitives, each with an integer coefficient, instantiated as ExpressionTerm.

#### 10.1.8.2 Attribute negate

**REQ. 223** **The attribute "negate" shall negate each of the terms in the Expression. It is the unary minus operator for the polynomials.**

`Expression::negate:Expression`

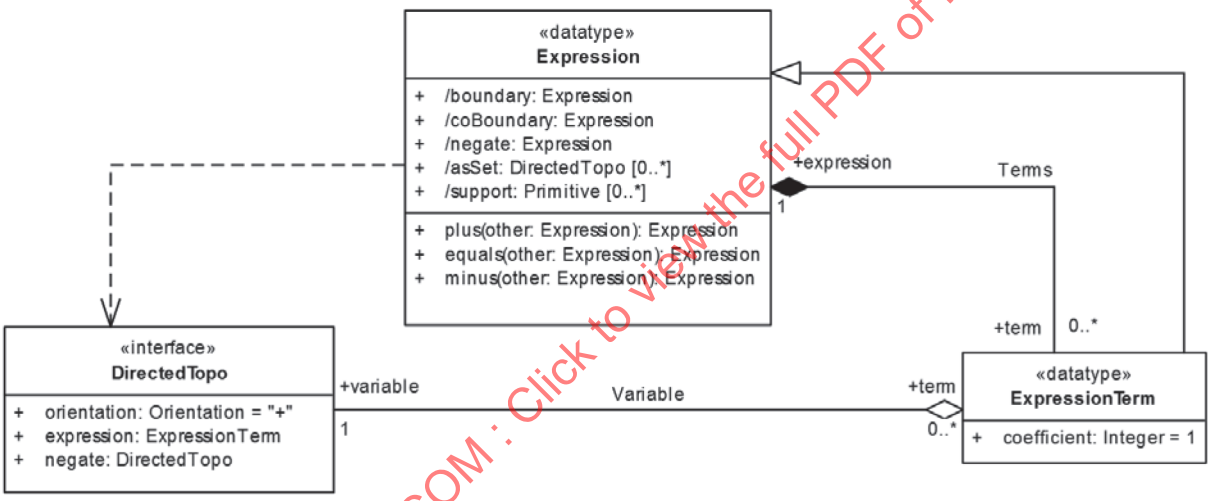



**Figure 39 — Expression**

#### 10.1.8.3 Attribute boundary

**REQ. 224** **The attribute "boundary" shall replace each Primitive in each DirectedTopo in this Expression with its boundary represented as sequence of DirectedTopo objects and shall simplify the resultant expression. Boundaries always consist of Primitives of one lower dimension.**

**REQ. 225** **If the dimension of all the Primitives in this Expression is zero (the Primitives are all nodes), then the boundary operation shall return a zero Expression (no terms).**

`Expression::boundary:Expression`

#### 10.1.8.4 Attribute coBoundary

**REQ. 226** **The attribute "coBoundary" shall replace each Primitive in each DirectedTopo in this Expression with its coBoundary and simplify the resultant expression.**

Coboundaries always consist of Primitives of one higher dimension. If the underlying geometry is not the boundary of anything, the coboundary is empty, which is zero as an expression. If the dimension of all the Primitives in this Expression is the same as the dimension of the corresponding maximal Complex, then the coBoundary operation will return a zero Expression.

```
Expression::coBoundary:Expression
```

#### 10.1.8.5 Attribute support

**REQ. 227** **The attribute "support" shall cast this Expression as a set of Primitives for use in calculating geometric operators.**

The operation is essentially the "asSet" operation followed by a traversal of the "Centre" association between DirectedTopo and Primitive. If the Expression has no terms after simplifications, then the support is the empty set "Ø".

```
Expression::support:Primitive[0..*]
```

#### 10.1.8.6 Attribute asSet

**REQ. 228** **The attribute "asSet" shall cast this Expression as a set of DirectedTopo for use in calculating geometric operators. The attribute "asSet" output shall include adding all boundary elements to the set until DirectedNodes are reached. The support of an Expression shall be a valid Complex.**

If the Expression has no terms after simplifications, then the support is the empty set "Ø".

```
Expression::asSet:DirectedTopo[0..*]
```

#### 10.1.8.7 Operation: plus(expression)

The operation "plus" acts as vector addition for Expressions.

**REQ. 229** **The operation "plus" shall combine DirectedTopo elements that have the same underlying instances of Primitive by adding their "orientation" coefficients. It shall remove any terms with zero-value coefficient.**

```
Expression::plus(other:Expression):Expression
```

#### 10.1.8.8 Operation: minus

**REQ. 230** **The operation "minus" shall act as vector subtraction for Expressions. It shall combine DirectedTopo elements that have the same underlying instances of Primitive by subtracting their "orientation" coefficients. It shall remove any terms with zero coefficients.**

```
Expression::minus(other:Expression):Expression
```

#### 10.1.8.9 Operation: equals

**REQ. 231** **The operation "equals" shall return TRUE for a polynomial equality.**

The order of the elements (terms) is not significant.

```
Expression::equals(other:Expression):Boolean
```

#### 10.1.9 Datatype ExpressionTerm

Expressions, like polynomials, consist of a set of terms, which consist of a variable and a coefficient. The expressions act like vectors with the DirectedTopo as basis vectors. The coefficients are usually integers in most applications.

```
ExpressionTerm= < coefficient:Integer= 1,variable:DirectedTopo
```

Arithmetic shall be consistent with normal vector manipulation.

### 10.2 Requirements Class Topology Root Data

**REQ. 232** **An implementation of the requirements class Topology Data shall support all datatypes in Topology.**

### 10.3 Requirements Class Node

#### 10.3.1 Semantics

The Node (see Figure 33) package contains all the primitives for one dimension and supports classes for representations of their structural relationships.

**REQ. 233** **An implementation of the Requirements Class Node shall implement Requirements Class Topology root.**

**REQ. 234** **An implementation of the Package Node shall have all instances and properties specified for this package, its contents and its dependencies, contained in the UML model for this package in this document.**

#### 10.3.2 Interface Node

##### 10.3.2.1 Semantics

Node (Figure 37) inherits all of its interfaces from Primitive, with some elaboration on the structure of boundary and coboundary.

**REQ. 235** **For Node, the "coBoundary" defined at Topology shall always return a set of references to DirectedEdges indicating which edges enter (positive DirectedEdges) and which leave (negative DirectedEdges) the node.**

NOTE In 2-dimensional maximal Complex containing this Node, the coBoundary can be sorted as a clockwise circular sequence in any geometric realization of this maximal Complex. In a 3D complex, the ordering is arbitrary.

```
Node::coBoundary:DirectedEdge[0..*]

Node::coBoundary.spoke:DirectedEdge[0..*]
```

##### 10.3.2.2 Association: Centre

Each Primitive, including Node, is associated with two DirectedTopo instances.



```
Node::proxy:DirectedNode[2]
```

In and of itself, a node has no orientation defined, but in terms of a boundary of an edge the usual formulae is:

```
Edge.boundary = endNode - startNode
```

This is an expression, and the two terms are represented by DirectedNodes. The endNode is a positive DirectedNode; the startNode is a negative DirectedNode - see 10.1.5 and 10.1.9.

#### 10.3.2.3 Attribute boundary

The boundary operation for Node shall overrides that defined at Topology by specifying the Empty set.

```
Node::boundary.isEmpty=TRUE
```

#### 10.3.2.4 Constraints

The Node's dimension shall be 0, and its boundary is empty.

```
Node:

    dimension=0;

    boundary.isEmpty=TRUE;
```

NOTE A node can still be isolated in a face and be the end of an edge, as long as that edge is not on the boundary of the containing face. The geometric realization of this would be a curve that dangles in space, but terminates at its intersection with a surface.

### 10.3.3 Interface DirectedNode

The class "DirectedNode" supports Node in the computational topology class Expression. For Node, the operation "boundary" defined at Topology shall always return a zero-valued expression, corresponding to empty geometry. This operation is overridden from Topology.

```
Node::boundary.isEmpty
```

## 10.4 Requirements Class Edge

### 10.4.1 Interface Edge

#### 10.4.1.1 Semantics

The primitive Edge (see Figure 33) is the 1-dimensional primitive for topology. For Edge, the operation "boundary" defined at Topology shall return a pair of nodes, one at the start of the edge (negative DirectedNode) and one at the end (positive DirectedNode). This operation is overridden from Topology. The same information may be represented as an association.

```
Edge::boundary:DirectedNode[2]
```

**REQ. 236** **An implementation of the Requirements Class Edge shall implement Requirements Class Node.**

**REQ. 237** **An implementation of the Package Edge shall have all instances and properties specified for this package, its contents and its dependencies, contained in the UML model for this package in this document.**

#### 10.4.1.2 Attribute coBoundary

For Edge, the operation "coBoundary" defined at Topology shall return a circular sequence of directed faces indicating which faces use this edge (positive DirectedFace) or its negative proxy (negative DirectedFace) on their boundary. The circular sequence shall represent a clockwise enumeration of these faces as viewed from the end of the associated curve in any geometric realization of the maximal Complex in which this Edge is contained. This operation is overridden from Topology. The same information may be implemented as an association.

```
Edge::coBoundary: DirectedFace[0..*]

Edge::coBoundary.spoke:DirectedFace[0..*]
```

NOTE In the 2-dimensional planar case, the coboundary has at most two faces. In the full topology case, there are precisely 2, one directed face having a positive "+" orientation and the associated face lying to the left of the edge, and the other directed face having a negative "–" orientation, and the associated face lying to the right of the edge.

#### 10.4.1.3 Attribute boundary

**REQ. 238** **The boundary operation for Edge shall overrides that defined at Topology by specifying a start node and end node. The Edge shall also have an association Boundary with association role boundary that specifies this same information as two directed edges, oriented positively for the end node and negatively for the start node.**

```
Edge::boundary:DirectedNode[2]
```

#### 10.4.1.4 Association: Centre

Each Primitive, including Edge is associated with two DirectedTopo instances.

```
Edge::proxy:DirectedEdge[2]

DirectedEdge::topo:Edge
```

NOTE In the 2-dimensional planar case, each directed edge bounds at most one face, precisely one face in a full planar topology. In the 3-dimensional case, or in a non-planar 2D complex, a directed edge can bound several faces.

#### 10.4.1.5 Constraints

The Edge shall have dimension 1.

```
Edge:dimension=1
```

### 10.4.2 Interface DirectedEdge

The interface "DirectedEdge" supports Edge in the computational topology class Expression. It is analogous to the concept of an OrientableCurve, in the sense that it acts as a proxy for the base curve/edge when needed.



## 10.5 Requirements Class Face

### 10.5.1 Semantics

The class "Face" (see Figure 33) provides topological primitives for Surface.

**REQ. 239** **An implementation of the Requirements Class Face shall implement Requirements Class Edge.**

**REQ. 240** **An implementation of the Package Face shall have all instances and properties specified for this package, its contents and its dependencies, contained in the UML model for this package in this document**

### 10.5.2 Interface Face

#### 10.5.2.1 Attribute boundary

For Face, the attribute "boundary" defined at Topology shall be a set of directed edges with appropriate orientation. This operation is overridden from Topology. The same information may be represented as an association.

```
Face::boundary:FaceBoundary
```

NOTE The same restriction on the meaning of exterior applies to the topology as did to the geometry.

**REQ. 241** **The Face shall also have an association Boundary with association role boundary that specifies this same information as directed edges, oriented positively for the left side of the edge and negatively for the right.**

```
Face::boundary:DirectedEdge[1..*]
```

The additional information that is included in the boundary is the organization of the Face boundary into rings and an indication as to which ring is the exterior.

#### 10.5.2.2 Attribute coBoundary

For Face, the attribute "coBoundary" defined at Topology shall be a set of references to directed solids indicating which solids use this face (positive DirectedSolid) or its negative proxy (negative DirectedSolid) in their boundary. This operation is overridden from Topology. The same information may be implemented as an association.

```
Face::coBoundary: DirectedSolid[0..2]

Face::coBoundary.spoke: DirectedSolid[0..2]
```

#### 10.5.2.3 Association Centre roles

Each Primitive, including Face is associated with two DirectedTopo instances.

```
Face::proxy:DirectedFace[2]

DirectedFace::topo: Face
```

#### 10.5.2.4 Constraints

**REQ. 242** **Face.dimension shall be two.**

```
Face:Topology::dimension=2
```

### 10.5.3 Interface DirectedFace

DirectedFaces shall be used in defining the boundary of a Solid.

## 10.6 Requirements Class Topology Solid

### 10.6.1 Interface Solid

#### 10.6.1.1 Semantics

The class "Solid" (Figure 33) provides topological primitives for Solid.

**REQ. 243** **An implementation of the Requirements Class Topology Solid shall implement Requirements Class Face.**

**REQ. 244** **An implementation of the Package Solid shall have all instances and properties specified for this package, its contents, and its dependencies, contained in the UML model for this package in this document.**

#### 10.6.1.2 Attribute boundary

For Solid, "boundary" defined at Topology shall return a collection of faces or their negative proxies.

**REQ. 245** **The Solid shall has a boundary consisting of directedFaces, oriented positively for below the face and negatively for above the face.**

```
Solid::boundary:DirectedFace[1..*]
```

The additional information that is returned by the boundary operator is the organization of the SolidBoundary into shells and an indication as to which shell is the exterior.

#### 10.6.1.3 Attribute coBoundary

For Solid, the operation "coBoundary" shall return NULL.

```
Solid::coBoundary:NULL
```

#### 10.6.1.4 Association role: Centre

Each Primitive, including Solid is associated with two DirectedTopo instances.

```
Solid::proxy[2]:DirectedSolid

DirectedSolid::topo:Solid
```