INTERNATIONAL STANDARD

# ISO 10303-42

First edition
1994-12-15



# Industrial automation systems and integration — Product data representation and exchange —

## Part 42:
Integrated generic resources: Geometric and topological representation

*Systèmes d'automatisation industrielle et intégration — Représentation et échange de données de produits —*

*Partie 42: Ressources génériques intégrées: Représentation géométrique et topologique*

Reference number
ISO 10303-42:1994(E)

# Contents







International Organization for Standardization
Case Postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

**Annexes**

**Figures**

**Tables**

## Foreword

The International Organization for Standardization (ISO) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards 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.

Draft International Standards adopted by technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75% of the member bodies casting a vote.

International Standard ISO 10303-42 was prepared by Technical Committee ISO/TC 184, *Industrial automation systems and integration*, Subcommittee SC4, *Industrial data and global manufacturing programming languages.*

ISO 10303 consists of the following parts under the general title *Industrial automation systems and integration – Product data representation and exchange:*

- Part 1, Overview and fundamental principles;
- Part 11, Description methods: The *EXPRESS* language reference manual;
- Part 21, Implementation methods: Clear text encoding of the exchange structure;
- Part 22, Implementation methods: Standard data access interface specification;
- Part 31, Conformance testing methodology and framework: General concepts;
- Part 32, Conformance testing methodology and framework: Requirements on testing laboratories and clients;
- Part 41, Integrated generic resources: Fundamentals of product description and support;
- Part 42, Integrated generic resources: Geometric and topological representation;
- Part 43, Integrated generic resources: Representation structures;
- Part 44, Integrated generic resources: Product structure configuration;
- Part 45, Integrated generic resources: Materials;
- Part 46, Integrated generic resources: Visual presentation;
- Part 47, Integrated generic resources: Shape variation tolerances;
- Part 49, Integrated generic resources: Process structure and properties;

- Part 101, Integrated application resources: Draughting;
- Part 104, Integrated application resources: Finite element analysis;
- Part 105, Integrated application resources: Kinematics;
- Part 201, Application protocol: Explicit draughting;
- Part 202, Application protocol: Associative draughting;
- Part 203, Application protocol: Configuration controlled design;
- Part 207, Application protocol: Sheet metal die planning and design;
- Part 210, Application protocol: Printed circuit assembly product design data;
- Part 213, Application protocol: Numerical control process plans for machined parts.

The structure of this International Standard is described in ISO 10303-1. The numbering of the parts of this International Standard reflects its structure:

- Part 11 specifies the description methods;
- Parts 21 and 22 specify the implementation methods;
- Parts 31 and 32 specify the conformance testing methodology and framework;
- Parts 41 to 49 specify the integrated generic resources;
- Parts 101 to 105 specify the integrated application resources;
- Parts 201 to 213 specify the application protocols.

Should further parts be published, they will follow the same numbering pattern.

Annexes A and B form an integral part of this part of ISO 10303. Annexes C, D, E are for information only.



**Diskette**

Users should note that this part of ISO 10303 comprises a diskette:
— the short names of entities given in annex A are also included on the diskette;
— the EXPRESS listings (annex C) are provided on the diskette only;
— a method to enable users to report errors in the documentation is given. Full details are provided in the file.

# Introduction

ISO 10303 is an International Standard for the computer-interpretable representation and exchange of product data. The objective is to provide a neutral mechanism capable of describing product data throughout the life cycle of a product independent from any particular system. The nature of this description makes it suitable not only for neutral file exchange, but also as a basis for implementing and sharing product databases and archiving.

This International Standard is organized as a series of parts, each published separately. The parts of ISO 10303 fall into one of the following series: description methods, integrated resources, application protocols, abstract test suites, implementation methods, and conformance testing. The series are described in ISO 10303–1. This part of ISO 10303 is a member of the integrated resources series. Major subdivisions of this International Standard are:

- Geometry
- Topology
- Geometric models

This part of ISO 10303 specifies the integrated resources used for geometric and topological representation. Their primary application is for explicit representation of the shape or geometric form of a product model. The shape representation presented here has been designed to facilitate stable and efficient communication when mapped to a physical file.

The geometry in clause 4 is exclusively the geometry of parametric curves and surfaces. It includes the curve and surface entities and other entities, functions and data types necessary for their definition. A common scheme has been used for the definition of both two-dimensional and three-dimensional geometry. All geometry is defined in a coordinate system which is established as part of the context of the item which it represents. These concepts are fully defined in ISO 10303 Part 43.

The topology in clause 5 is concerned with connectivity relationships between objects rather than with the precise geometric form of objects. This clause contains the basic topological entities and specialised subtypes of these. In some cases the subtypes have geometric associations. Also included are functions, particularly constraint functions, and data types necessary for the definitions of the topological entities.

The geometric models in clause 6 provide basic resources for the communication of data describing the precise size and shape of three-dimensional solid objects. The geometric shape models provide a complete representation of the shape which in many cases includes both geometric and topological data. Included here are the two classical types of solid model, constructive solid geometry (CSG) and boundary representation (B-rep). Other entities, providing a rather less complete description of the geometry of a product, and with less consistency constraints, are also included.

This page intentionally left blank

# Industrial automation systems and integration — Product data representation and exchange — Part 42 : Integrated generic resources: Geometric and topological representation

## 1 Scope

This part of ISO 10303 specifies the resource constructs for the explicit geometric and topological representation of the shape of a product. The scope is determined by the requirements for the explicit representation of an ideal product model; tolerances and implicit forms of representation in terms of features are out of scope. The geometry in clause 4 and the topology in clause 5 are available for use independently and are also extensively used by the various forms of geometric shape model in clause 6. In addition, this part of ISO 10303 specifies specialisations of the concepts of representation where the elements of representation are geometric.

### 1.1 Geometry

The following are within the scope of the geometry schema:

- definition of points, vectors, parametric curves and parametric surfaces;
- definition of transformation operators;
- points defined directly by their coordinate values or in terms of the parameters of an existing curve or surface;
- definition of conic curves and elementary surfaces;
- definition of curves defined on a parametric surface;
- definition of general parametric spline curves and surfaces;
- definition of point, curve and surface replicas;
- definition of offset curves and surfaces;
- definition of intersection curves.

The following are outside the scope of this part of ISO 10303:

- all other forms of procedurally defined curves and surfaces;

– curves and surfaces which do not have a parametric form of representation;

– any form of explicit representation of a ruled surface.

NOTE – For a ruled surface the geometry is critically dependent upon the parametrisation of the boundary curves and the method of associating pairs of points on the two curves. A ruled surface with B-spline boundary curves can however be exactly represented by the B-spline surface entity.

## 1.2 Topology

The following are within the scope of the topology schema:

– definition of the fundamental topological entities vertex, edge, and face, each with a specialised subtype to enable it to be associated with the geometry of a point, curve, or surface, respectively;

– collections of the basic entities to form topological structures of path, loop and shell and constraints to ensure the integrity of these structures;

– orientation of topological entities.

## 1.3 Geometric Shape Models

The following are within the scope of the geometric model schema:

– data describing the precise geometric form of three-dimensional solid objects;

– constructive solid geometry (CSG) models;

– definition of CSG primitives and half-spaces;

– creation of solid models by sweeping operations;

– manifold boundary representation (B-rep) models;

– constraints to ensure the integrity of B-rep models;

– surface models;

– wireframe models;

– geometric Sets;

– creation of a replica of a solid model in a new location.

The following are outside the scope of this part of ISO 10303:

– non-manifold boundary representation models;

– spatial occupancy forms of solid models (such as octree models);

– assemblies and mechanisms.

## 2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO 10303.. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO 10303. are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 8824-1:–[1)], *Information Technology – Open Sysyems Interconnection – Abstract Syntax Notation One (ASN.1) – Part 1: Specification of Basic Notation.*

ISO 10303-1:1994, *Industrial automation systems and integration – Product data representation and exchange – Part 1: Overview and fundamental principles.*

ISO 10303-11:1994, *Industrial automation systems and integration – Product data representation and exchange – Part 11: Description methods: The EXPRESSLanguage Reference Manual.*

ISO 10303-41:1994, *Industrial automation systems and integration – Product data representation and exchange – Part 41 : Integrated generic resources: Fundamentals of product description and support.*

ISO 10303-43:1994, *Industrial automation systems and integration – Product data representation and exchange – Part 43 : Integrated generic resources: Representation structures.*

## 3 Definitions, symbols and abbreviations

### 3.1 Definitions

For the purposes of this part of ISO 10303, the following definitions apply.

**3.1.1 arcwise connected:** an entity is arcwise connected if any two arbitrary points in its domain can be connected by a curve that lies entirely within the domain.

**3.1.2 axi-symmetric:** an entity is axi-symmetric if it has an axis of symmetry such that the object is invariant under all rotations about this axis.

1) To be published.

**3.1.3 bounds:** the topological entities of lower dimensionality which mark the limits of a topological entity. The bounds of a face are loops, and the bounds of an edge are vertices.

**3.1.4 boundary:** the set of mathematical points $x$ in a domain $X$ contained in $R^m$ for which there is an open ball $U$ in $R^m$ containing $x$ such that the intersection $U \cap X$ is homeomorphic to an open set in the closed $d$ -dimensional half-space $R^d_+$, for some $d \leq m$, where the homeomorphism carries $x$ into the origin in $R^d_+$.

NOTES

1 – $R^d_+$ is defined to be the set of all mathematical points $(x_1, ..., x_d)$ in $R^d$ with $x_1 \geq 0$.

2 – For this purpose, the word "open" has its usual mathematical meaning. It does not relate to "open surface" as defined elsewhere in this part of ISO 10303.

**3.1.5 boundary representation solid model (Brep):** a type of geometric model in which the size and shape of the solid is defined in terms of the faces, edges and vertices which make up its boundary.

**3.1.6 closed curve:** a curve such that both end points are the same.

**3.1.7 closed surface:** a connected 2-manifold that divides space into exactly two connected components, one of which is finite.

**3.1.8 completion of a topological entity:** a set consisting of the entity in question together with all the faces, edges and vertices referenced, directly or indirectly, in the definition of the bounds of that entity.

**3.1.9 connected:** equivalent to **arcwise connected**.

**3.1.10 connected component:** a maximal connected subset of a domain.

**3.1.11 constructive solid geometry (CSG):** a type of geometric modelling in which a solid is defined as the result of a sequence of regularised Boolean operations operating on solid models.

**3.1.12 coordinate space:** a reference system that associates a unique set of $n$ parametesrs with each point in an $n$-dimensional space.

**3.1.13 curve:** a set of mathematical points which is the image, in two- or three-dimensional space, of a continuous function defined over a connected subset of the real line ($R^1$), and which is not a single point.

**3.1.14 cycle:** a chain of alternating vertices and edges in a graph such that the first and last vertices are the same.

**3.1.15** ***d*-manifold with boundary:** a domain which is the union of its *d*-dimensional interior and its boundary.

**3.1.16 dimensionality:** the number of independent coordinates in the parameter space of a geometric entity. The dimensionality of topological entities which need not have domains is specified in the entity definitions. The dimensionality of a list or set is the maximum of the dimensionalities of the elements of that list or set.

**3.1.17 domain:** the mathematical point set in model space corresponding to an entity.

**3.1.18 euler equations:** Equations used to verify the topological consistency of objects. Various equalities relating topological properties of entities are derived from the invariance of a number known as the Euler characteristic. Typically, these are used as quick checks on the integrity of the topological structure. A violation of an Euler condition signals an "impossible" object. Two special cases are important in this document. The Euler equation for graphs is discussed in 5.2.3. Euler conditions for surfaces are discussed in 5.4.23 and 5.4.25.

**3.1.19 extent:** the measure of the content of the domain of an entity, measured in units appropriate to the dimensionality of the entity. Thus, length, area and volume are used for dimensionalities 1, 2, and 3, respectively. Where necessary, the symbol $\Xi$ will be used to denote extent.

**3.1.20 finite:** an entity is finite (sometimes called bounded) if there is a finite upper bound on the distance between any two points in its domain.

**3.1.21 genus of a graph:** the integer-valued invariant defined algorithmically by the graph traversal algorithm described in the note in 5.2.3.

**3.1.22 genus of a surface:** the number of handles that must be added to a sphere to produce a surface homeomorphic to the surface in question.

**3.1.23 geometrically founded:** a property of **geometric_representation_items** asserting their relationship to a coordinate space in which the coordinate values of points and directions on which they depend for position and orientation are measured.

**3.1.24 geometrically related:** the relationship between two **geometric_representation_items** in the same context by which the concepts of distance and direction between them are defined.

**3.1.25 geometric coordinate system:** the underlying global rectangular Cartesian coordinate system to which all geometry refers.

**3.1.26 graph:** a set of vertices and edges. The graphs discussed in this document are generally called pseudographs in the technical literature because they allow self-loops and also multiple edges connecting the same two vertices.

**3.1.27 handle:** the structure distinguishing a torus from a sphere, which can be viewed as a cylindrical tube connecting two holes in a surface.

**3.1.28 homeomorphic:** domains $X$ and $Y$ are homeomorphic if there is a continuous function $f$ from $X$ to $Y$ which is a one-to-one correspondence, so that the inverse function $f^{-1}$ exists, and if $f^{-1}$ is also continuous.

**3.1.29 inside:** domain $X$ is inside domain $Y$ if both domains are contained in the same Euclidean space, $R^m$, and $Y$ separates $R^m$ into exactly two connected components, one of which is finite, and $X$ is contained in the finite component.

**3.1.30 interior:** the $d$-dimensional interior of a $d$-dimensional domain $X$ contained in $R^m$ is the set of mathematical points $x$ in $X$ for which there is an open ball $U$ in $R^m$ containing $x$ such that the intersection $U \cap X$ is homeomorphic to an open ball in $R^d$.

**3.1.31 list:** an ordered homogeneous collection with possibly duplicate members. A list is represented by an enclosing pair of brackets, i.e. $[A]$.

**3.1.32 model space:** a space with dimensionality 2 or 3 in which the geometry of a physical object is defined.

**3.1.33 open curve:** a curve which has two distinct end points.

**3.1.34 open surface:** a surface which is a manifold with boundary, but is not closed. I.e., either it is not finite, or it does not divide space into exactly two connected components.

**3.1.35 orientable:** a surface is orientable if a consistent, continuously varying choice can be made of the sense of the normal vectors to the surface.

NOTE – This does not require a continuously varying choice of the *values* of the normal vectors; the surface may have tangent plane discontinuities.

**3.1.36 overlap:** two entities overlap when they have shells, faces, edges, or vertices in common.

**3.1.37 parameter range:** the range of valid parameter values for a curve or surface.

**3.1.38 parameter space:** the one-dimensional space associated with a curve via its uniquely defined parametrisation or the two-dimensional space associated with a surface.

**3.1.39 placement coordinate system:** a rectangular Cartesian coordinate system associated with the placement of a geometric entity in space, used to describe the interpretation of the attributes and to associate a unique parametrisation with curve and surface entities.

**3.1.40 self-intersect:** a curve or surface self-intersects if there is a mathematical point in its domain which is the image of at least two points in the object's parameter range, and one of those two points lies in the interior of the parameter range. A vertex, edge or face self-intersects if its domain does.

NOTE – A curve or surface is not considered to be self-intersecting just because it is closed.

**3.1.41 self-loop:** an edge that has the same vertex at both ends.

**3.1.42 set:** an unordered collection in which no two members are equal.

**3.1.43 space dimensionality:** the number of parameters required to define the location of a point in the coordinate space.

**3.1.44 surface:** a set of mathematical points which is the image of a continuous function defined over a connected subset of the plane ($R^2$).

**3.1.45 topological sense:** the sense of a topological entity as derived from the order of its attributes.

EXAMPLES

1 – The topological sense of an edge is from the edge start vertex to the edge end vertex.

2 – The topological sense of a path follows the edges in their listed order.

## 3.2 Symbols

For the purposes of this part of ISO 10303, the following symbols and definitions apply.

### 3.2.1 Geometry and mathematical symbology

The mathematical symbol convention used in the geometry schema is given in table 1.

### 3.2.2 Topology symbols

An attempt has been made to define precisely the constraints that shall be met by the topological entities. In many cases these are defined symbolically. This subclause describes the notation used for this purpose. It should be noted that the definitions given here are independent of EXPRESS definitions and usage.

The topological constructs are **vertex**, **edge**, **path**, **loop**, **face** (and **subface**) and **shell**. These will be referred to by the following symbols $V, E, P, L, F$ and $S$, respectively.

Some of these entities take particular forms and a superscript is used to distinguish between these forms if necessary.

EXAMPLE 3 – A **loop** may be a **vertex_loop**, an **edge_loop** or a **poly_loop**. These forms are denoted as $L^v, L^e, L^p$.

**Table 1 – Geometry mathematical symbology**

| Symbol | Definition |
|---|---|
| $a$ | Scalar quantity |
| $\mathbf{A}$ | Vector quantity |
| $\langle\rangle$ | Vector normalisation |
| $\mathbf{a}$ | Normalised vector (e.g. $\mathbf{a} = \langle\mathbf{A}\rangle = \mathbf{A}/\lvert\mathbf{A}\rvert$) |
| $\times$ | Vector (cross) product |
| $\cdot$ | Scalar product |
| $\mathbf{A} \rightarrow \mathbf{B}$ | $\mathbf{A}$ is transformed to $\mathbf{B}$ |
| $\boldsymbol{\lambda}(u)$ | Parametric curve |
| $\mathcal{C}(x, y, z)$ | Analytic curve |
| $\boldsymbol{\sigma}(u, v)$ | Parametric surface |
| $\mathcal{S}(x, y, z)$ | Analytic surface |
| $\mathcal{C}_x$ | Partial differential of $\mathcal{C}$ with respect to $x$ |
| $\boldsymbol{\sigma}_u$ | Partial derivative of $\boldsymbol{\sigma}(u, v)$ with respect to $u$ |
| $\mathcal{S}_x$ | Partial derivative of $\mathcal{S}$ with respect to $x$ |
| $\lvert\,\rvert$ | Absolute value, or magnitude or determinant |
| $R^m$ | m-dimensional real space |

Table 2 lists the symbols used in the topology schema.

An undirected edge is an entity of type **edge** which is not of the subtype **oriented_edge**. In some instances of the entity definitions, a topological attribute may take the form of a (topological + logical) pair, this is generally represented by the oriented subtype. A subscript is used to distinguish between the topological and the (topological + logical) pairing. For example, $E$ and $E_l$ or $S^o$ and $S^o_l$.

Several topological entities use an Orientation Flag to indicate whether the direction of a referenced entity agrees with or is opposed to the direction of the referencing entity. If the Flag is TRUE, the direction of the referenced entity is correct but if the Flag is FALSE, the direction of the referenced entity should be (conceptually) reversed. It can happen that there are several Orientation Flags in the chain of entities from the high-level referencing entity to the low-level referenced entity. The direction of a low-level entity with respect to a high-level entity is obtained by evaluating the *not exclusive or* ($\odot$) of the chain of Orientation Flags. For example, a Face references a Loop + Loopflag, a Loop references an Edge + Edgeflag and an Edge references a Curve + Curveflag. The Face's "FaceCurveflag" is given by

$$\text{FaceCurveflag} = \text{Loopflag} \odot \text{Edgeflag} \odot \text{Curveflag}$$

where *not exclusive or* is interpreted as true if the two flags have the same value and is defined by the truth table:

$$\begin{aligned} T \odot T &= T \\ T \odot F &= F = F \odot T \\ F \odot F &= T. \end{aligned}$$

**Table 2 – Topology Symbol Definitions**

| Symbol | Definition |
|---|---|
| $V$ | Vertex |
| $\mathcal{V}$ | Number of unique vertices |
| $E$ | Undirected edge |
| $\mathcal{E}$ | Number of unique undirected edges |
| $E_l$ | Oriented edge |
| $\mathcal{E}_l$ | Number of unique oriented edges |
| $G^e$ | Edge genus |
| $P$ | Path |
| $\mathcal{P}$ | Number of unique paths |
| $G^p$ | Path genus |
| $L$ | Loop |
| $\mathcal{L}$ | Number of unique loops |
| $L_l$ | Face bound |
| $\mathcal{L}_l$ | Number of unique face bounds |
| $L^e$ | Edge loop |
| $L^p$ | Poly loop |
| $L^v$ | Vertex loop |
| $G^l$ | Loop genus |
| $F$ | Face |
| $\mathcal{F}$ | Number of unique faces |
| $H^f$ | Face genus |
| $S$ | Shell |
| $\mathcal{S}$ | Number of unique shells |
| $S^c$ | Closed shell |
| $S^o$ | Open shell |
| $S^v$ | Vertex shell |
| $S^w$ | Wire shell |
| $H^s$ | Shell genus |
| $\Xi$ | Extent |
| $\{A\}$ | Set of entities of type $A$ |
| $[A]$ | List of entities of type $A$ |

Thus

$$F \odot T \odot F = T.$$

## 3.3 Abbreviations

For the purposes of this part of ISO 10303, the following abbreviations apply.

**B-rep:** boundary representation solid model;

**CSG:** constructive solid geometry.

# 4 Geometry

The following EXPRESS declaration begins the **geometry_schema** and identifies the necessary external references.

<u>*EXPRESS* specification</u>:

```
*)
SCHEMA geometry_schema;
  REFERENCE FROM representation_schema
    (representation,
     functionally_defined_transformation,
     representation_item,
     representation_context,
     definitional_representation,
     item_in_context,
     using_representations);
  REFERENCE FROM measure_schema
    (length_measure,
     positive_length_measure,
     plane_angle_measure,
     plane_angle_unit,
     positive_plane_angle_measure,
     parameter_value,
     global_unit_assigned_context);
  REFERENCE FROM topology_schema
    (edge_curve,
     face_surface,
     poly_loop,
     vertex_point);
   REFERENCE FROM geometric_model_schema
     (solid_model,
      boolean_result,
      sphere,
      right_circular_cone,
      right_circular_cylinder,
      torus,
      block,
      right_angular_wedge,
      half_space_solid,
      shell_based_surface_model,
      face_based_surface_model,
      shell_based_wireframe_model,
      edge_based_wireframe_model,
      geometric_set);
(*
```

NOTES

1 – The schemas referenced above can be found in:

| | |
|---|---|
| representation_schema | ISO 10303-43 |
| measure_schema | ISO 10303-41 |
| topology_schema | clause 5 of this part of ISO 10303 |
| geometric_model_schema | clause 6 of this part of ISO 10303 |

2 – The references to **topology_schema** and to **geometric_model_schema** are required only for the definition of the **geometric_representation_item** supertype.

3 – See annex D, figures D.1 to D.12, for a graphical presentation of this schema.



## 4.1 Introduction

The subject of the **geometry_schema** is the geometry of parametric curves and surfaces. The **representation_schema** (see ISO 10303-43) and the **geometric_representation_context** defined in this Part of ISO 10303, provide the context in which the geometry is defined. This enables a distinction to be made between those items which are geometrically related and those existing in independent coordinate spaces. In particular, each **geometric_representation_item** has a **geometric_representation_context** which includes as an attribute the Euclidean dimension of its coordinate space. The coordinate system for this space is referred to as the geometric coordinate system in this clause. Units associated with **length_measures** and **plane_angle_measures** are assumed to be assigned globally within this context. A global rule (**compatible_dimension**) ensures that all **geometric_representation_items** in the same **geometric_representation_context** have the same space dimensionality. The space dimensionality **dim** is an inherited derived attribute of all subtypes of **geometric_representation_item**.

## 4.2 Fundamental concepts and assumptions

### 4.2.1 Space dimensionality

All geometry shall be defined in a right-handed rectangular Cartesian coordinate system with the same units on each axis. A common scheme has been used for the definition of both two-dimensional and three-dimensional geometry. Points and directions exist in both a two-dimensional and a three-dimensional form; these forms are distinguished solely by the presence, or absence, of a third coordinate value. Complex geometric entities are all defined using points and directions from which their space dimensionality can be deduced.

### 4.2.2 Geometric relationships

All **geometric_representation_items** which are included as **items** in a **representation** having a **geometric_representation_context** are geometrically related. Any such **geometric_representation_item** is said to be geometrically founded in the context of that **representation**. No geometric relationship, such as distance between points, is assumed to exist for **geometric_representation_items** occurring as **items** in different **representations**.

### 4.2.3 Parametrisation of analytic curves and surfaces

Each curve or surface specified here has a defined parametrisation. In some instances the definitions are in parametric terms. In others, the conic curves and elementary surfaces, the definitions are in geometric terms.

In this latter case a placement coordinate system is used to define the parametrisation. The geometric definitions contain some, but not all, of the data required for this. The relevant data to define this placement coordinate system is contained in the **axis2_placement** associated with the individual curve and surface entities.

### 4.2.4 Curves

The curve entities defined in 4.4 include lines, elementary conics, a general parametric polynominal curve, and some referentially or procedurally defined curves. All the curves have a well defined parametrisation which makes it possible to trim a curve or identify points on the curve by parameter value. For the conic curves, a method of representation is used which separates their geometric form from their orientation and position in space. In each case, the position and orientation information is conveyed by an **axis2_placement**. The general purpose parametric curve is represented by the **b_spline_curve** entity. This was selected as the most stable form of representation for the communication of all types of polynomial and rational parametric curves. With appropriate attribute values and subtypes, a **b_spline_curve** entity is capable of representing single span or spline curves of explicit polynomial, rational, Bézier or B-spline type. A **composite_curve** entity, which includes the facility to communicate continuity information at the curve-to-curve transition points, is provided for the construction of more complex curves.

The **offset_curve** and **curve_on_surface** types are curves defined with reference to other geometry. Separate **offset_curve** entities exist for 2D and 3D applications. The curve on surface entities include an **intersection_curve** which represents the intersection of two surfaces. Such a curve may be represented in 3D space or in the 2D parameter space of either of the surfaces.

### 4.2.5 Surfaces

The surface entities support the requirements of simple boundary representation (B-rep) solid modelling system and enable the communication of general polynomial and rational parametric surfaces. The simple surfaces are the planar, spherical, cylindrical, conical and toroidal surfaces, a **surface_of_revolution** and a **surface_of_linear_extrusion**. As with curves, all surfaces have an associated standard parametrisation. In many cases the surfaces, as defined, are unbounded; it is assumed that they will be bounded either explicitly or implicitly. Explicit bounding is achieved with the **rectangular_trimmed_surface** or **curve_bounded_surface** entities; implicit bounding requires the association of additional topological information to define a **face**.

The **b_spline_surface** entity and its subtypes provide the most general capability for the communication of all types of polynomial and rational biparametric surfaces. This entity uses control points as the most stable form of representation for the surface geometry. The **offset_surface** entity is intended for the communication of a surface obtained as a simple normal offset from a given surface. capability to connect together a rectangular mesh of distinct surface patches, specifying the degree of continuity from patch to patch.

### 4.2.6 Preferred form

Some of the geometric entities provide the potential capability of defining an item of geometry in more than one way. Such multiple representations are accommodated by requiring the nomination of a 'preferred form' or 'master representation'. This is the form which is used to determine the parametrisation.

NOTE – The **master_representation** attribute acknowledges the impracticality of ensuring that multiple forms are indeed identical and allows the indication of a preferred form. This would probably be determined by the creator of the data. All characteristics, such as parametrisation, domain, and results of evaluation, for an entity having multiple representations, are derived from the master representation. Any use of the other representations is a compromise for practical considerations.



## 4.3 geometry_schema type definitions

### 4.3.1 dimension_count

A **dimension_count** is a positive integer used to define the coordinate space dimensionality of a **geometric_representation_context**.

*EXPRESS* specification:

```
*)
TYPE dimension_count = INTEGER;
WHERE
  WR1: SELF > 0;
END_TYPE;
(*
```

Formal propositions:

**WR1:** A **dimension_count** shall be positive.

### 4.3.2 transition_code

This type conveys the continuity properties of a composite curve or surface. The continuity referred to is geometric, not parametric continuity.

*EXPRESS* specification:

```
*)
TYPE transition_code = ENUMERATION OF
  (discontinuous,
   continuous,
   cont_same_gradient,
   cont_same_gradient_same_curvature);
END_TYPE;
(*
```

Enumerated item definitions:

**discontinuous:** The segments, or patches, do not join. This is permitted only at the boundary of the curve or surface to indicate that it is not closed.

**continuous:** The segments, or patches, join, but no condition on their tangents is implied.

**cont_same_gradient:** The segments, or patches, join, and their tangent vectors, or tangent planes, are parallel and have the same direction at the joint; equality of derivatives is not required.

**cont_same_gradient_same_curvature:** For a curve, the segments join, their tangent vectors are parallel and in the same direction, and their curvatures are equal at the joint; equality of derivatives is not required. For a surface this implies that the principal curvatures are the same and that the principal directions are coincident along the common boundary.

### 4.3.3 preferred_surface_curve_representation

This type is used to indicate the preferred form of representation for a surface curve, which is either a curve in geometric space or in the parametric space of the underlying surfaces.

*EXPRESS* specification:

```
*)
TYPE preferred_surface_curve_representation = ENUMERATION OF
  (curve_3d,
   pcurve_s1,
   pcurve_s2);
END_TYPE;
(*
```

Enumerated item definitions:

**curve_3d:** The curve in three-dimensional space is preferred.

**pcurve_s1:** The first pcurve is preferred.

**pcurve_s2:** The second pcurve is preferred.

### 4.3.4 b_spline_curve_form

This type is used to indicate a particular geometric form represented by the B-spline curve.

*EXPRESS* specification:

```
*)
TYPE b_spline_curve_form = ENUMERATION OF
  (polyline_form,
   circular_arc,
   elliptic_arc,
```

```
   parabolic_arc,
   hyperbolic_arc,
   unspecified);
END_TYPE;
(*
```

Enumerated item definitions:

**polyline_form:** A connected sequence of line segments represented by degree 1 B-spline basis functions.

**circular_arc:** An arc of a circle, or a complete circle represented by a B-spline curve.

**elliptic_arc:** An arc of an ellipse, or a complete ellipse, represented by a B-spline curve.

**parabolic_arc:** An arc of finite length of a parabola represented by a B-spline curve.

**hyperbolic_arc:** An arc of finite length of one branch of a hyperbola represented by a B-spline curve.

**unspecified:** A B-spline curve for which no particular form is specified.

### 4.3.5 b_spline_surface_form

This type is used to indicate that the B-spline surface represents a part of a surface of some specific form.

*EXPRESS* specification:

```
*)
TYPE b_spline_surface_form = ENUMERATION OF
  (plane_surf,
   cylindrical_surf,
   conical_surf,
   spherical_surf,
   toroidal_surf,
   surf_of_revolution,
   ruled_surf,
   generalised_cone,
   quadric_surf,
   surf_of_linear_extrusion,
   unspecified);
END_TYPE;
(*
```

Enumerated item definitions:

**plane_surf:** A bounded portion of a plane represented by a B-spline surface of degree 1 in each parameter.

**cylindrical_surf:** A bounded portion of a cylindrical surface.

**conical_surf:** A bounded portion of the surface of a right circular cone.

**spherical_surf:** A bounded portion of a sphere, or a complete sphere, represented by a B-spline surface.

**toroidal_surf:** A torus, or portion of a torus, represented by a B-spline surface.

**surf_of_revolution:** A bounded portion of a surface of revolution.

**ruled_surf:** A surface constructed from two parametric curves by joining with straight lines corresponding points with the same parameter value on each of the curves.

**generalised_cone:** A special case of a ruled surface in which the second curve degenerates to a single point; when represented by a B-spline surface all the control points along one edge will be coincident.

**quadric_surf:** A bounded portion of one of the class of surfaces of degree 2 in the variables x, y and z.

**surf_of_linear_extrusion:** A bounded portion of a surface of linear extrusion represented by a B-spline surface of degree 1 in one of the parameters.

**unspecified:** A surface for which no particular form is specified.

### 4.3.6 knot_type

This type indicates that the B-spline knots shall have a particularly simple form enabling the knots themselves to be defaulted.

For details of the interpretation of these types see the B-spline curve entity definition (4.4.29).

*EXPRESS* specification:

```
*)
TYPE knot_type = ENUMERATION OF
  (uniform_knots,
   unspecified,
   quasi_uniform_knots,
   piecewise_bezier_knots);
END_TYPE;
  (*
```

Enumerated item definitions:

**uniform_knots:** The form of knots appropriate for a uniform B-spline curve.

**unspecified:** The type of knots is not specified. This includes the case of non uniform knots.

**quasi_uniform_knots:** The form of knots appropriate for a quasi-uniform B-spline curve.

**piecewise_bezier_knots:** The form of knots appropriate for a piecewise Bézier curve.

### 4.3.7 extent_enumeration

This type is used to describe the quantitive extent of an object.

*EXPRESS* specification:

```
*)
TYPE extent_enumeration = ENUMERATION OF
  (invalid,
   zero,
   finite_non_zero,
   infinite);
END_TYPE;
(*
```

Enumerated item definitions:

**invalid:** The concept of extent is not valid for the quantity being measured.

**zero:** The extent is zero.

**finite_non_zero:** The extent is finite (bounded) but not zero.

**infinite:** The extent is not finite.

### 4.3.8 trimming_preference

This type is used to indicate the preferred way of trimming a parametric curve where the trimming is multiply defined.

*EXPRESS* specification:

```
*)
TYPE trimming_preference = ENUMERATION OF
  (cartesian, parameter,
   unspecified);
END_TYPE;
(*
```

Enumerated item definitions:

**cartesian:** Indicates that trimming by cartesian point is preferred.

**parameter:** Indicates a preference for the parameter value.

**unspecified:** Indicates that no preference is communicated.

### 4.3.9 axis2_placement

This select type collects together both versions of the axis2 placement as used in two-dimensional or in three-dimensional Cartesian space. This enables entities requiring this information to reference them without specifying the space dimensionality.

*EXPRESS* specification:

```
*)
TYPE axis2_placement = SELECT
  (axis2_placement_2d,
   axis2_placement_3d);
END_TYPE;
(*
```

### 4.3.10 curve_on_surface

A **curve_on_surface** is a curve on a parametric surface. It may be any of the following

- a **pcurve** or
- a **surface_curve**, including the specialised subtypes of **intersection_curve** and **seam_curve**, or
- a **composite_curve_on_surface**.

The **curve_on_surface** select type collects these curves together for reference purposes.

*EXPRESS* specification:

```
*)
TYPE curve_on_surface = SELECT
  (pcurve,
   surface_curve,
   composite_curve_on_surface);
END_TYPE;
(*
```

### 4.3.11 pcurve_or_surface

This select type enables a surface curve to identify as an attribute the associated surface or pcurve.

*EXPRESS* specification:

```
*)
TYPE pcurve_or_surface = SELECT
  (pcurve,
   surface);
END_TYPE;
(*
```

### 4.3.12 trimming_select

This select type identifies the two possible ways of trimming a parametric curve, by a cartesian point on the curve, or by a REAL number defining a parameter value within the parametric range of the curve.

*EXPRESS* specification:

```
*)
TYPE trimming_select = SELECT
  (cartesian_point,
   parameter_value);
END_TYPE;
(*
```

### 4.3.13 vector_or_direction

This type is used to identify the types of entity which can participate in vector computations.

*EXPRESS* specification:

```
*)
TYPE vector_or_direction = SELECT
  (vector,
   direction);
END_TYPE;
(*
```

## 4.4 geometry_schema entity definitions

This subclause contains all the explicit geometric entities. Except for entities defined in a parameter space, all geometry is defined in a right-handed cartesian coordinate system (the geometric coordinate system). The space dimensionality of this coordinate system is established by the context of the **geometric_representation_item**. The curve and surface definitions are all given essentially in terms of points and/or vectors and/or scalar (length) values.

### 4.4.1 geometric_representation_context

A **geometric_representation_context** is a **representation_context** in which **geometric_representation_items** are geometrically founded.

A **geometric_representation_context** is a distinct coordinate space, spatially unrelated to other coordinate spaces except as those coordinate spaces are specifically related by an appropriate transformation. (See 3.1 for definitions of geometrically founded and coordinate space.)

*EXPRESS* specification:

```
*)
ENTITY geometric_representation_context
  SUBTYPE OF (representation_context);
  coordinate_space_dimension : dimension_count;
END_ENTITY;
(*
```

Attribute definitions:

**coordinate_space_dimension:** The integer **dimension_count** of the coordinate space which is the **geometric_representation_context**.

NOTE – Any constraints on the allowed range of **coordinate_space_dimension** are outside the scope of this part of ISO 10303.

### 4.4.2 geometric_representation_item

A **geometric_representation_item** is a **representation_item** that has the additional meaning of having geometric position or orientation or both. This meaning is present by virtue of:

– being a **cartesian_point** or a **direction**;

– referencing directly a **cartesian_point** or a **direction**;

– referencing indirectly a **cartesian_point** or a **direction**.

NOTE 1 – An indirect reference to a **cartesian_point** or **direction** means that a given **geometric_representation_item** references the **cartesian_point** or **direction** through one or more intervening attributes. In many cases this information is given in the form of an **axis2_placement**.

EXAMPLES

4 – Consider a circle. It gains its geometric position and orientation by virtue of a reference to **axis2_placement** that in turn references a **cartesian_point** and several **directions**.

5 – A **manifold_solid_brep** is a **geometric_representation_item** that through several layers of **topological_representation_items**, references **curves**, **surfaces** and **points**. Through additional intervening entities curves and surfaces reference **cartesian_point** and **direction**.

NOTES

2 – The intervening entities, which are all of type **representation_item**, need not be of subtype **geometric_representation_item**. Consider the **manifold_solid_brep** from the above example. One of the intervening levels of **representation_item** is a **closed_shell**. This is a **topological_representation_item** and does not require a **geometric_representation_context** in its own right. When used as part of the definition of a **manifold_solid_brep** that itself is a **geometric_representation_item**, it is founded in a **geometric_representation_context**.

3 – A **geometric_representation_item** inherits the need to be related to a **representation_context** in a **representation**. The rule **compatible_dimension** ensures that the **representation_context** is a **geometric_representation_context**. When in the context of geometry, this relationship causes the **geometric_representation_item** to be geometrically founded.

*EXPRESS* specification:

```
*)
ENTITY geometric_representation_item
  SUPERTYPE OF (ONEOF(point, direction, vector, placement,
                      cartesian_transformation_operator, curve, surface,
                      edge_curve, face_surface, poly_loop, vertex_point,
                      solid_model, boolean_result, sphere, right_circular_cone,
                      right_circular_cylinder, torus, block,
                      right_angular_wedge, half_space_solid,
                      shell_based_surface_model, face_based_surface_model,
                      shell_based_wireframe_model, edge_based_wireframe_model,
                      geometric_set))
  SUBTYPE OF (representation_item);
DERIVE
  dim : dimension_count := dimension_of(SELF);
WHERE
  WR1: SIZEOF (QUERY (using_rep <* using_representations (SELF) |
       NOT ('GEOMETRY_SCHEMA.GEOMETRIC_REPRESENTATION_CONTEXT' IN
       TYPEOF (using_rep.context_of_items)))) = 0;
END_ENTITY;
(*
```

Attribute definitions:

**dim:** The coordinate **dimension_count** of the **geometric_representation_item**.

NOTES

4 – The **dim** attribute is derived from the **coordinate_space_dimension** of a **geometric_representation_context** in which the **geometric_representation_item** is geometrically founded.

5 – A **geometric_representation_item** is geometrically founded in one or more **geometric_representation_contexts**, all of which have the same **coordinate_space_dimension**. See the rule **compatible_dimension** in 4.5.1.

Formal propositions:

**WR1:** The context of any representation referencing a **geometric_representation_item** shall be of the type **geometric_representation_context**.

### 4.4.3 point

A **point** is a location in some real Cartesian coordinate space $R^m$, for $m = 1, 2$ or 3.

*EXPRESS* specification:

```
*)
ENTITY point
  SUPERTYPE OF (ONEOF(cartesian_point, point_on_curve, point_on_surface,
                      point_replica, degenerate_pcurve))
  SUBTYPE OF (geometric_representation_item);
END_ENTITY;
(*
```

### 4.4.4 cartesian_point

A **cartesian_point** is a **point** defined by its coordinates in a rectangular Cartesian coordinate system, or in a parameter space. The entity is defined in a one, two or three-dimensional space as determined by the number of coordinates in the list.

NOTE – For the purposes of defining geometry in this part of ISO 10303 only two or three-dimensional points are used.

*EXPRESS* specification:

```
*)
ENTITY cartesian_point
  SUBTYPE OF (point);
  coordinates  : LIST [1:3] OF length_measure;
END_ENTITY;
(*
```

Attribute definitions:

**coordinates[1]:** The first coordinate of the **point** location.

**coordinates[2]:** The second coordinate of the **point** location; this will not exist in the case of a one-dimensional point.

**coordinates[3]:** The third coordinate of the **point** location; this will not exist in the case of a one or two-dimensional point.

**SELF\geometric_representation_item.dim:** The dimensionality of the space in which the **point** is defined. This is an inherited derived attribute from the geometric representation item supertype and for a cartesian point is determined by the number of coordinates in the list.



### 4.4.5 point_on_curve

A **point_on_curve** is a **point** which lies on a **curve**. The point is determined by evaluating the **curve** at a specific parameter value. The coordinate space dimensionality of the point is that of the **basis_curve**.

*EXPRESS* specification:

```
*)
ENTITY point_on_curve
  SUBTYPE OF (point);
  basis_curve     : curve;
  point_parameter : parameter_value;
END_ENTITY;
(*
```

Attribute definitions:

**basis_curve:** The **curve** to which **point_parameter** relates.

**point_parameter:** The parameter value of the **point** location.

**SELF\geometric_representation_item.dim:** The dimensionality of the space in which the **point_on_curve** is defined. This is the same as that of the **basis_curve**.

Informal propositions:

**IP1:** The value of the **point_parameter** shall not be outside the parametric range of the **curve**.

### 4.4.6 point_on_surface

A **point_on_surface** is a point which lies on a parametric surface. The point is determined by evaluating the surface at a particular pair of parameter values.

*EXPRESS* specification:

```
*)
ENTITY point_on_surface
  SUBTYPE OF (point);
  basis_surface     : surface;
  point_parameter_u : parameter_value;
  point_parameter_v : parameter_value;
END_ENTITY;
(*
```

Attribute definitions:

**basis_surface:** The **surface** to which the parameter values relate.

**point_parameter_u:** The first parameter value of the **point** location.

**point_parameter_v:** The second parameter value of the **point** location.

**SELF\geometric_representation_item.dim:** The dimensionality of the coordinate space of the **point_on_surface**. This is the same as that of the **basis_surface**.

Informal propositions:

**IP1:** The parametric values specified for u and v shall not be outside the parametric range of the **basis_surface**.

### 4.4.7 point_replica

This defines a replica of an existing point (the parent) in a different location. The replica has the same coordinate space dimensionality as the parent point.

*EXPRESS* specification:

```
*)
ENTITY point_replica
  SUBTYPE OF (point);
  parent_pt      : point;
  transformation : cartesian_transformation_operator;
WHERE
  WR1: transformation.dim = parent_pt.dim;
```

```
  WR2: acyclic_point_replica (SELF,parent_pt);
END_ENTITY;
(*
```

Attribute definitions:

**parent_pt:** The point to be replicated.

**transformation:** The Cartesian transformation operator which defines the location of the point replica.

Formal propositions:

**WR1:** The coordinate space dimensionality of the transformation attribute shall be the same as that of the **parent_pt**.

**WR2:** A **point_replica** shall not participate in its own definition.

### 4.4.8 degenerate_pcurve

A **degenerate_pcurve** is an entity with the structure of a pcurve, but which in three-dimensional model space collapses to a single point. It is thus a subtype of point, not of curve.

NOTE – For example, the apex of a cone could be represented as a degenerate pcurve.

*EXPRESS* specification:

```
*)
ENTITY degenerate_pcurve
  SUBTYPE OF (point);
  basis_surface:  surface;
  reference_to_curve : definitional_representation;
WHERE
  WR1: SIZEOF(reference_to_curve\representation.items) = 1;
  WR2: 'GEOMETRY_SCHEMA.CURVE' IN TYPEOF
                  (reference_to_curve\representation.items[1]);
  WR3: reference_to_curve\representation.
                 items[1]\geometric_representation_item.dim =2;
END_ENTITY;
(*
```



Attribute definitions:

**basis_surface:** The surface on which the **basis_curve** lies.

**reference_to_curve:** The association of the **pcurve** and the parameter space curve which degenerates to the (equivalent) point.

Formal propositions:

**WR1:** The set of items in the **definitional_representation** entity corresponding to the **reference_to_curve** shall have exactly one element.

**WR2:** The unique item in the set shall be a curve.

**WR3:** The dimensionality of this parameter space curve shall be 2.

Informal propositions:

**IP1:** Regarded as a curve in model space, the **degenerate_pcurve** shall have zero arc length.

### 4.4.9 evaluated_degenerate_pcurve

This entity represents the result of evaluating a degenerate pcurve and associates it with a Cartesian point.

*EXPRESS* specification:

```
*)
ENTITY evaluated_degenerate_pcurve
  SUBTYPE OF (degenerate_pcurve);
  equivalent_point : cartesian_point;
END_ENTITY;
(*
```

Attribute definitions:

**equivalent_point:** The point in the geometric coordinate system represented by the degenerate pcurve.

### 4.4.10 direction

This entity defines a general direction vector in two or three dimensional space. The actual magnitudes of the components have no effect upon the direction being defined, only the ratios x:y:z or x:y are significant.

NOTE – The components of this entity are not normalised. If a unit vector is required it should be normalised before use.

*EXPRESS* specification:

```
*)
ENTITY direction
  SUBTYPE OF (geometric_representation_item);
  direction_ratios : LIST [2:3] OF REAL;
WHERE
  WR1: SIZEOF(QUERY(tmp <* direction_ratios | tmp <> 0.0)) > 0;
END_ENTITY;
(*
```



Attribute definitions:

**direction_ratios[1]:** The component in the direction of the X axis.

**direction_ratios[2]:** The component in the direction of the Y axis.

**direction_ratios[3]:** The component in the direction of the Z axis; this will not be present in the case of a direction in two-dimensional space.

**SELF\geometric_representation_item.dim:** The coordinate space dimensionality of the direction. This is an inherited attribute of the **geometric_representation_item** supertype; for this entity it is determined by the number of **direction_ratios** in the list.

Formal propositions:

**WR1:** The magnitude of the direction vector shall be greater than zero.

### 4.4.11 vector

This entity defines a vector in terms of the direction and the magnitude of the vector. The value of the **magnitude** attribute defines the magnitude of the vector.

> NOTE – The magnitude of the vector must not be calculated from the components of the **orientation** attribute. This form of representation was selected to reduce problems with numerical instability. For example a vector of magnitude 2.0 mm and equally inclined to the coordinate axes could be represented with orientation attribute of (1.0,1.0,1.0).

*EXPRESS* specification:

```
*)
ENTITY vector
  SUBTYPE OF (geometric_representation_item);
  orientation : direction;
  magnitude   : length_measure;
WHERE
  WR1 : magnitude >= 0.0;
END_ENTITY;
(*
```

Attribute definitions:

**orientation:** The direction of the **vector**.

**magnitude:** The magnitude of the **vector**. All vectors of **magnitude** 0.0 are regarded as equal in value regardless of the **orientation** attribute.

**SELF\geometric_representation_item.dim:** The dimensionality of the space in which the **vector** is defined.

Formal propositions:

**WR1:** The magnitude shall be positive or zero.



### 4.4.12 placement

A **placement** locates a geometric item with respect to the coordinate system of its geometric context. It locates the item to be defined and, in the case of the axis placement subtypes, gives its orientation.

*EXPRESS* specification:

```
*)
ENTITY placement
```

```
   SUPERTYPE OF (ONEOF(axis1_placement,axis2_placement_2d,axis2_placement_3d))
   SUBTYPE OF (geometric_representation_item);
   location : cartesian_point;
 END_ENTITY;
 (*
```

Attribute definitions:

**location:** The geometric position of a reference point, such as the centre of a circle, of the item to be located.

### 4.4.13 axis1_placement

The direction and location in three-dimensional space of a single axis. An **axis1_placement** is defined in terms of a locating point (inherited from the placement supertype) and an axis direction; this is either the direction of **axis** or defaults to (0.0,0.0,1.0). The actual direction for the axis placement is given by the derived attribute **z**.

*EXPRESS* specification:

```
 *)
 ENTITY axis1_placement
   SUBTYPE OF (placement);
   axis      : OPTIONAL direction;
 DERIVE
   z         : direction := NVL(normalise(axis), direction([0.0,0.0,1.0]));
 WHERE
   WR1: SELF\geometric_representation_item.dim  = 3;
 END_ENTITY;
 (*
```

Attribute definitions:

**SELF\placement.location:** A reference point on the axis.

**axis:** The direction of the local Z axis.

**z:** The normalised direction of the local Z axis.

**SELF\geometric_representation_item.dim:** The space dimensionality of the **axis1_placement**, which is determined from its **location**, and is always equal to 3.

Formal propositions:

**WR1:** The coordinate space dimensionality shall be 3.

### 4.4.14 axis2_placement_2d

The location and orientation in two-dimensional space of two mutually perpendicular axes. An **axis2_placement_2d** is defined in terms of a point, (inherited from the placement supertype), and an axis. It can be used to locate and orientate an object in two-dimensional space and to define a placement coordinate system. The entity includes a point which forms the origin of the placement coordinate system. A direction vector is required to complete the definition of the placement coordinate system. The **ref_direction** defines the placement X axis direction; the placement Y axis direction is derived from this.

*EXPRESS* specification:

```
*)
ENTITY axis2_placement_2d
  SUBTYPE OF (placement);
  ref_direction : OPTIONAL direction;
DERIVE
  p              : LIST [2:2] OF direction := build_2axes(ref_direction);
WHERE
  WR1: SELF\geometric_representation_item.dim = 2;
END_ENTITY;
(*
```

Attribute definitions:

**SELF\placement.location:** The spatial position of the reference point which defines the origin of the associated placement coordinate system.

**ref_direction:** The direction used to determine the direction of the local X axis. If **ref_direction** is omitted, this direction is taken from the geometric coordinate system.

**p:** The axis set for the placement coordinate system.

**p[1]:** The normalised direction of the placement X axis. This is (1.0,0.0) if **ref_direction** is omitted.

**p[2]:** The normalised direction of the placement Y axis. This is a derived attribute and is orthogonal to **p[1]**.

Formal propositions:

**WR1:** The space dimensionality of the **axis2_placement_2d** shall be 2.

### 4.4.15 axis2_placement_3d

The location and orientation in three-dimensional space of two mutually perpendicular axes. An **axis2_placement_3d** is defined in terms of a point, (inherited from the placement supertype), and two (ideally orthogonal) axes. It can be used to locate and orientate a non axi-symmetric object in space and to define a placement coordinate system. The entity includes a point which forms the origin of the placement coordinate system. Two direction vectors are required to complete the definition of the placement coordinate system. The **axis** is the placement Z axis direction and the **ref_direction** is an approximation to the placement X axis direction.

NOTE – Let **z** be the placement Z axis direction and **a** be the approximate placement X axis direction. There are two methods, mathematically identical but numerically different, for calculating the placement X and Y axis directions.

a) The vector **a** is projected onto the plane defined by the origin point **P** and the vector **z** to give the placement X axis direction as $\mathbf{x} = \langle \mathbf{a} - (\mathbf{a} \cdot \mathbf{z})\mathbf{z} \rangle$. The placement Y axis direction is then given by $\mathbf{y} = \langle \mathbf{z} \times \mathbf{x} \rangle$.

b) The placement Y axis direction is calculated as $\mathbf{y} = \langle \mathbf{z} \times \mathbf{a} \rangle$ and then the placement X axis direction is given by $\mathbf{x} = \langle \mathbf{y} \times \mathbf{z} \rangle$.

The first method is likely to be the more numerically stable of the two, and is used here.

A placement coordinate system referenced by the parametric equations is derived from the **axis2_placement_3d** data for conic curves and elementary surfaces.

*EXPRESS* specification:

```
*)
ENTITY axis2_placement_3d
  SUBTYPE OF (placement);
  axis          : OPTIONAL direction;
  ref_direction : OPTIONAL direction;
DERIVE
  p             : LIST [3:3] OF direction := build_axes(axis,ref_direction);
WHERE
  WR1: SELF\placement.location.dim = 3;
  WR2: (NOT (EXISTS (axis))) OR (axis.dim = 3);
  WR3: (NOT (EXISTS (ref_direction))) OR (ref_direction.dim = 3);
  WR4: (NOT (EXISTS (axis))) OR (NOT (EXISTS (ref_direction))) OR
       (cross_product(axis,ref_direction).magnitude > 0.0);
END_ENTITY;
(*
```

Attribute definitions:

**SELF\placement.location:** The spatial position of the reference point and origin of the associated placement coordinate system.

**axis:** The exact direction of the local Z axis.

**ref_direction:** The direction used to determine the direction of the local X axis. If necessary an adjustment is made to maintain orthogonality to the **axis** direction. If **axis** and/or **ref_direction** is omitted, these directions are taken from the geometric coordinate system.

**p:** The axes for the placement coordinate system. The directions of these axes are derived from the attributes, with appropriate default values if required.

**p[1]:** The normalised direction of the local X axis.

**p[2]:** The normalised direction of the local Y axis

**p[3]:** The normalised direction of the local Z axis.

NOTE – See figure 1 for interpretation of attributes.

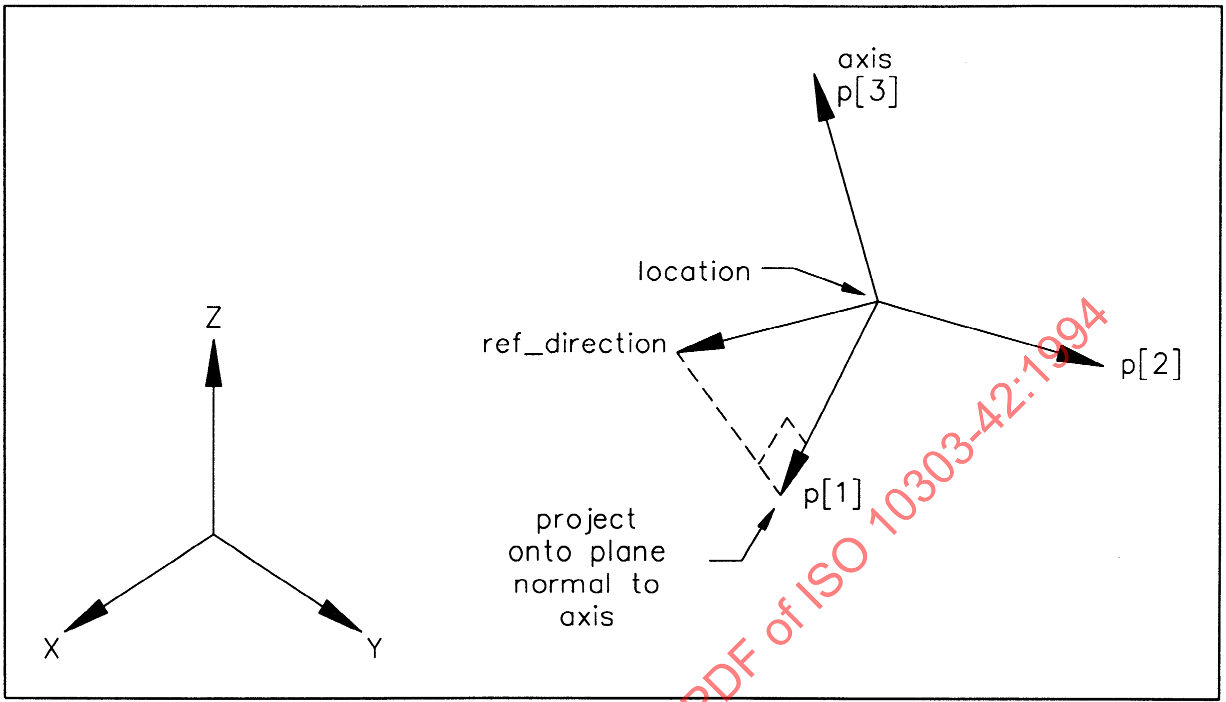


**Figure 1 – Axis2 placement 3D**



Formal propositions:

**WR1:** The space dimensionality of the **SELF\placement.location** shall be 3.

**WR2:** The space dimensionality of **axis** shall be 3.

**WR3:** The space dimensionality of **ref_direction** shall be 3.

**WR4:** The **axis** and the **ref_direction** shall not be parallel or anti-parallel. (This is required by the **build_axes** function.)

## 4.4.16 cartesian_transformation_operator

A **cartesian_transformation_operator** defines a geometric transformation composed of translation, rotation, mirroring and uniform scaling.

The list of normalised vectors **u** defines the columns of an orthogonal matrix **T**. These vectors are computed, by the **base_axis** function, from the direction attributes **axis1**, **axis2** and, in **cartesian_transformation_operator_3d**, **axis3**. If $|\mathbf{T}| = -1$, the transformation includes mirroring. The local origin point **A**, the scale value $S$ and the matrix **T** together define a transformation.

The transformation for a **point** with position vector **P** is defined by

$$\mathbf{P} \rightarrow \mathbf{A} + S\mathbf{T}\mathbf{P}$$

The transformation for a **direction d** is defined by

$$\mathbf{d} \rightarrow \mathbf{Td}$$

The transformation for a **vector** with orientation **d** and magnitude $k$ is defined by

$$\mathbf{d} \rightarrow \mathbf{Td}$$

and

$$k \rightarrow Sk$$

For those entities whose attributes include an **axis2_placement**, the transformation is applied, after the derivation, to the derived attributes **p** defining the placement coordinate **directions**. For a transformed **surface**, the direction of the surface normal at any point is obtained by transforming the normal, at the corresponding point, to the original **surface**. For geometric entities with attributes (such as the radius of a circle) which have the dimensionality of length, the values will be multiplied by $S$.

For curves on surface the **p_curve.reference_to_curve** will be unaffected by any transformation.

The **cartesian_transformation_operator** shall only be applied to geometry defined in a consistent system of units with the same units on each axis. With all attributes omitted, the transformation defaults to the identity transformation. The **cartesian_transformation_operator** shall only be instantiated as one of its subtypes.

NOTE – See figures 2(a-c) for demonstration of effect of transformation.

*EXPRESS* specification:

```
*)
ENTITY cartesian_transformation_operator
  SUPERTYPE OF(ONEOF(cartesian_transformation_operator_2d,
                              cartesian_transformation_operator_3d))
  SUBTYPE OF (geometric_representation_item,
                            functionally_defined_transformation);
  axis1        : OPTIONAL direction;
  axis2        : OPTIONAL direction;
  local_origin : cartesian_point;
  scale        : OPTIONAL REAL;
DERIVE
  scl          : REAL := NVL(scale, 1.0);
WHERE
  WR1: scl > 0.0;
END_ENTITY;
(*
```

Attribute definitions:

**axis1:** The direction used to determine **u**[1], the derived X axis direction.

**axis2:** The direction used to determine **u**[2], the derived Y axis direction.

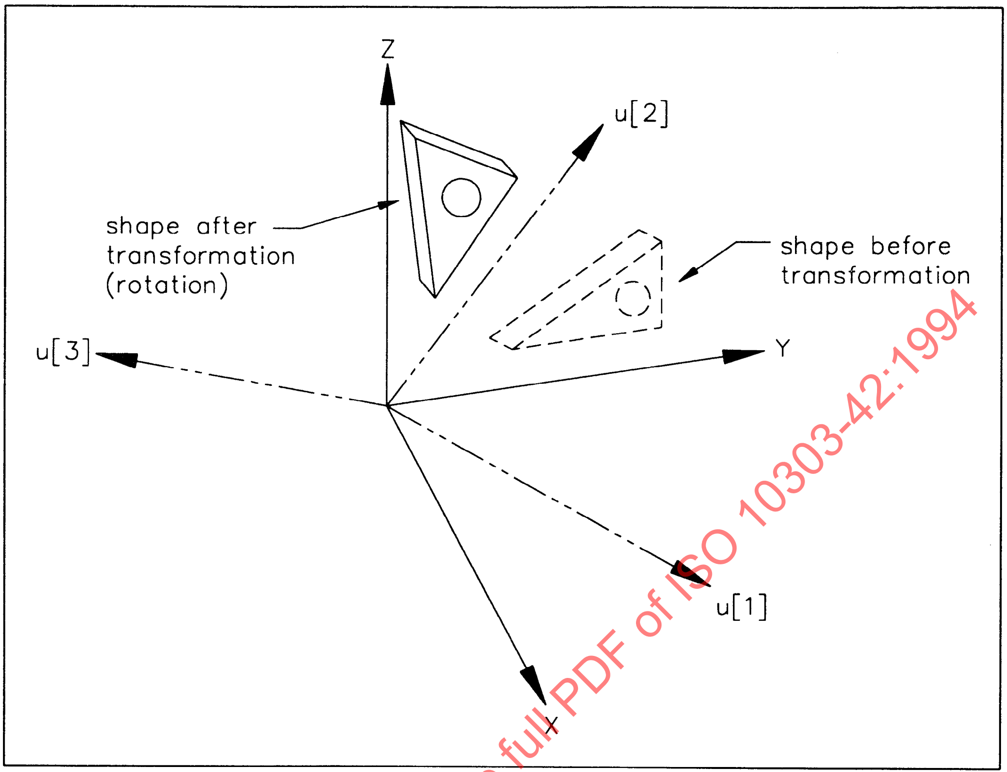


**Figure 2 – (a) Cartesian transformation operator 3D**



**local_origin:** The required translation, specified as a cartesian point. The actual translation included in the transformation is from the geometric origin to the local origin.

**scale:** The scaling value specified for the transformation.

**scl:** The derived scale $S$ of the transformation, equal to **scale** if that exists, or 1.0 otherwise.

Formal propositions:

**WR1:** The derived scaling **scl** shall be greater than zero.

## 4.4.17 cartesian_transformation_operator_3d

A **cartesian_transformation_operator_3d** defines a geometric transformation in three-dimensional space composed of translation, rotation, mirroring and uniform scaling.

The list of normalised vectors **u** defines the columns of an orthogonal matrix **T**. These vectors are computed from the direction attributes axis1, axis2 and axis3 by the **base_axis** function. If $|\mathbf{T}| = -1$, the transformation includes mirroring.

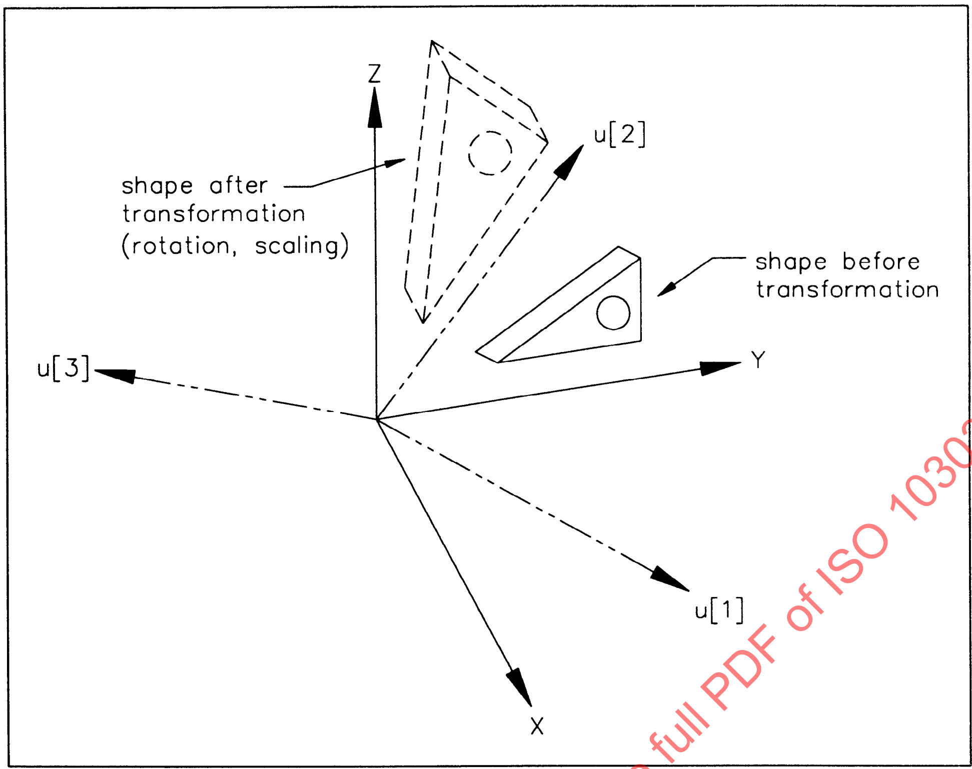


**Figure 2 – (b) Cartesian transformation operator 3D**



*EXPRESS* specification:

```
*)
ENTITY cartesian_transformation_operator_3d
  SUBTYPE OF (cartesian_transformation_operator);
  axis3 : OPTIONAL direction;
DERIVE
  u      : LIST[3:3] OF direction
         := base_axis(3,SELF\cartesian_transformation_operator.axis1,
                        SELF\cartesian_transformation_operator.axis2,axis3);
WHERE
  WR1: SELF\cartesian_transformation_operator.dim = 3;
END_ENTITY;
(*
```

Attribute definitions:

**SELF\cartesian_transformation_operator.axis1:** The direction used to determine **u**[**1**], the derived X axis direction. If necessary, **u**[**1**] is adjusted to make it orthogonal to **u**[**3**].

**SELF\cartesian_transformation_operator.axis2:** The direction used to determine **u**[**2**], the derived Y axis direction. If necessary, **u**[**2**] is adjusted to make it orthogonal to **u**[**1**] and **u**[**3**].

**SELF\cartesian_transformation_operator.axis3:** The exact direction of **u**[**3**], the derived

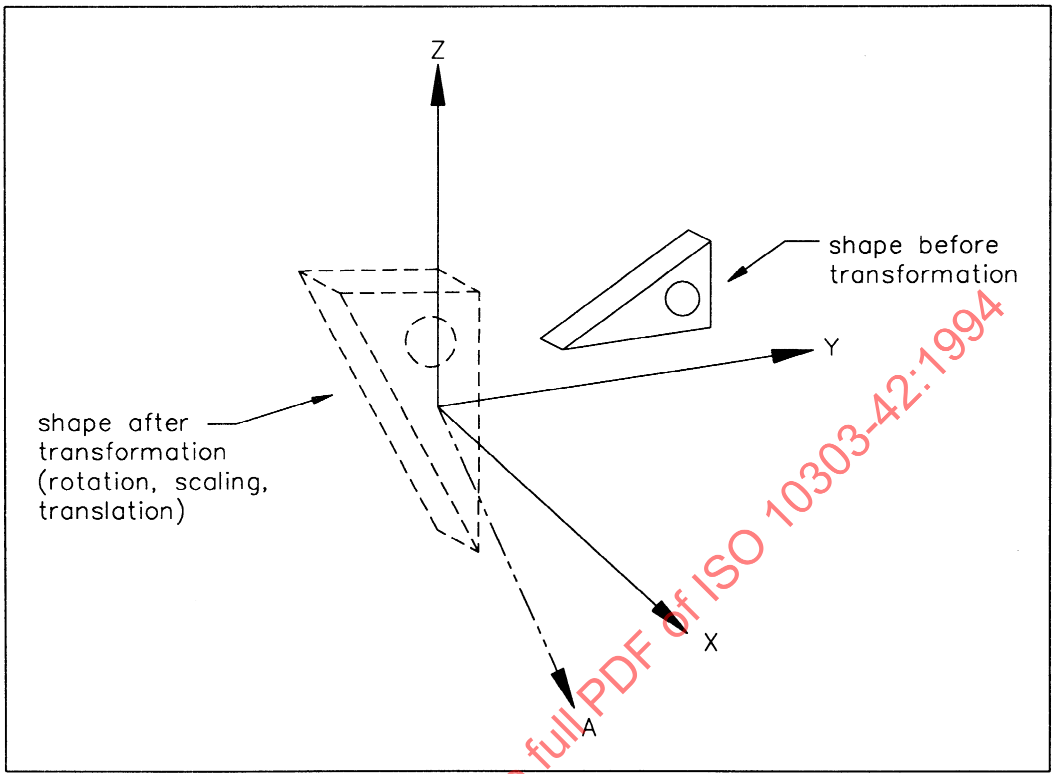


**Figure 2 – (c) Cartesian transformation operator**



Z axis direction.

**SELF\cartesian_transformation_operator.local_origin:** The required translation, specified as a cartesian point. The actual translation included in the transformation is from the geometric origin to the local origin.

**SELF\cartesian_transformation_operator.scale:** The scaling value specified for the transformation.

**SELF\cartesian_transformation_operator.scl:** The derived scale **S** of the transformation, equal to **scale** if that exists, or 1.0 otherwise.

**u:** The list of mutually orthogonal, normalised vectors defining the transformation matrix **T**. They are derived from the explicit attributes **axis3**, **axis1**, and **axis2** in that order.

Formal propositions:

**WR1:** The coordinate space dimensionality of this entity shall be 3.

### 4.4.18 cartesian_transformation_operator_2d

A **Cartesian_transformation_operator_2d** defines a geometric transformation in two-dimensional space composed of translation, rotation, mirroring and uniform scaling.

The list of normalised vectors **u** defines the columns of an orthogonal matrix **T**. These vectors are computed from the direction attributes axis1, axis2 by the **base_axis** function. If $|\mathbf{T}| = -1$, the transformation includes mirroring.

*EXPRESS* specification:

```
*)
ENTITY cartesian_transformation_operator_2d
  SUBTYPE OF (cartesian_transformation_operator);
DERIVE
  u : LIST[2:2] OF direction :=
      base_axis(2,SELF\cartesian_transformation_operator.axis1,
                  SELF\cartesian_transformation_operator.axis2,?);
WHERE
  WR1: SELF\cartesian_transformation_operator.dim = 2;
END_ENTITY;
(*
```



Attribute definitions:

**SELF\cartesian_transformation_operator.axis1:** The direction used to determine **u**[1], the derived X axis direction.

**SELF\cartesian_transformation_operator.axis2:** The direction used to determine **u**[2], the derived Y axis direction.

**SELF\cartesian_transformation_operator.local_origin:** The required translation, specified as a cartesian point. The actual translation included in the transformation is from the geometric origin to the local origin.

**SELF\cartesian_transformation_operator.scale:** The scaling value specified for the transformation.

**SELF\cartesian_transformation_operator.scl:** The derived scale **S** of the transformation, equal to **scale** if that exists, or 1.0 otherwise.

**u:** The list of mutually orthogonal, normalised vectors defining the transformation matrix **T**. They are derived from the explicit attributes **axis1**, and **axis2** in that order.

Formal propositions:

**WR1:** The coordinate space dimensionality of this entity shall be 2.

### 4.4.19 curve

A **curve** can be envisioned as the path of a point moving in its coordinate space.

*EXPRESS* specification:

```
*)
ENTITY curve
  SUPERTYPE OF (ONEOF(line, conic, pcurve, surface_curve,
                      offset_curve_2d, offset_curve_3d, curve_replica))
  SUBTYPE OF (geometric_representation_item);
END_ENTITY;
(*
```

Informal propositions:

**IP1:** A **curve** shall be arcwise connected.

**IP2:** A **curve** shall have an arc length greater than zero.

### 4.4.20 line

A line is an unbounded curve with constant tangent direction. A **line** is defined by a **point** and a **direction**. The positive direction of the line is in the direction of the **dir** vector.

The curve is parametrised as follows:

$$\begin{aligned}\mathbf{P} &= \text{pnt}\\ \mathbf{V} &= \text{dir}\\ \boldsymbol{\lambda}(u) &= \mathbf{P} + u\mathbf{V}\end{aligned}$$

and the parametric range is $-\infty < u < \infty$.

*EXPRESS* specification:

```
*)
ENTITY line
  SUBTYPE OF (curve);
  pnt : cartesian_point;
  dir : vector;
WHERE
  WR1: dir.dim  = pnt.dim;
END_ENTITY;
(*
```

Attribute definitions:

**pnt:** The location of the **line**.

**dir:** The direction of the **line**; the magnitude and units of **dir** affect the parametrisation of the line.

**SELF\geometric_representation_item.dim:** The dimensionality of the coordinate space for the **line**. This is an inherited attribute from the geometric representation item supertype.

Formal propositions:

**WR1: Pnt** and **dir** shall both be 2D or both be 3D entities.

## 4.4.21 conic

A **conic** is a planar curve which could be produced by intersecting a plane with a cone.

A **conic** curve is defined in terms of its intrinsic geometric properties rather than being described in terms of other geometry.

A **conic** entity always has a placement coordinate system defined by **axis2_placement**; the parametric representation is defined in terms of this placement coordinate system.

*EXPRESS* specification:

```
*)
ENTITY conic
  SUPERTYPE OF (ONEOF(circle, ellipse, hyperbola, parabola))
  SUBTYPE OF (curve);
  position: axis2_placement;
END_ENTITY;
(*
```

Attribute definitions:

**position:** The location and orientation of the conic. Further details of the interpretation of this attribute are given for the individual subtypes.



## 4.4.22 circle

A **circle** is defined by a radius and the location and orientation of the circle. Interpretation of the data shall be as follows:

$$\begin{aligned} \mathbf{C} &= \text{position.location (centre)} \\ \mathbf{x} &= \text{position.p[1]} \\ \mathbf{y} &= \text{position.p[2]} \\ \mathbf{z} &= \text{position.p[3]} \\ R &= \text{radius} \end{aligned}$$

and the circle is parametrised as

$$\boldsymbol{\lambda}(u) = \mathbf{C} + R((\cos u)\mathbf{x} + (\sin u)\mathbf{y})$$

The parametrisation range is $0 \le u \le 360$ degrees.

In the placement coordinate system defined above, the circle is the equation $\mathcal{C} = 0$, where

$$\mathcal{C}(x, y, z) = x^2 + y^2 - R^2$$

The positive sense of the circle at any point is in the tangent direction, $\mathbf{T}$, to the curve at the point, where

$$\mathbf{T} = (-\mathcal{C}_y, \mathcal{C}_x, 0).$$

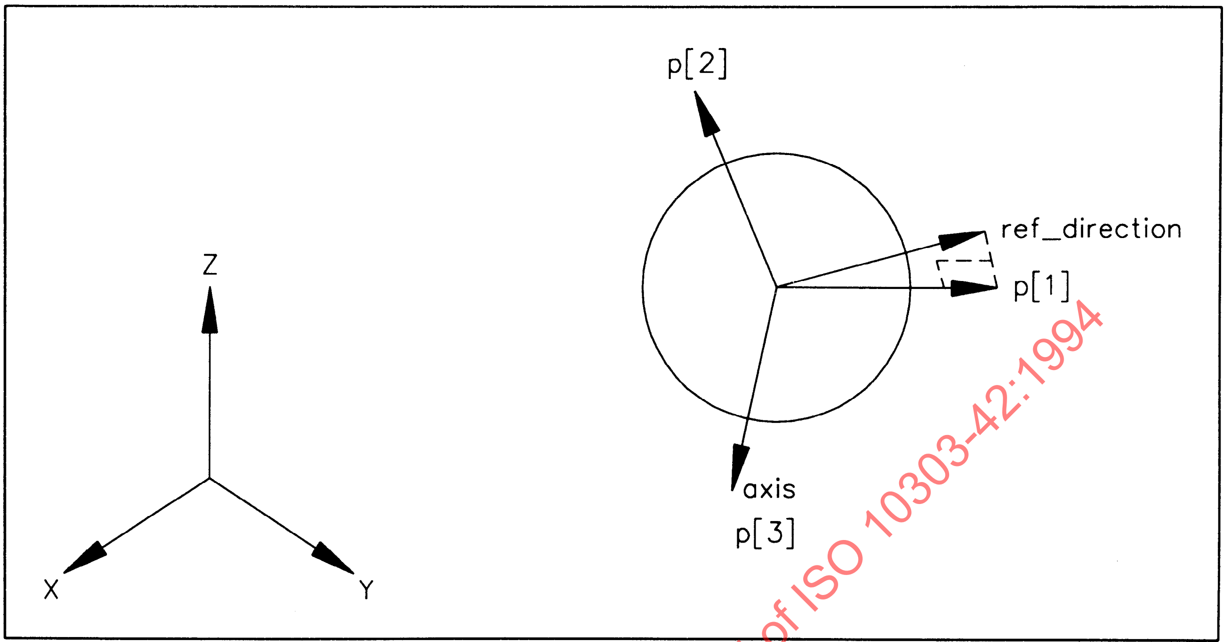


**Figure 3 – Circle**



NOTE – A circular arc is defined by using the **trimmed_curve** entity in conjunction with the **circle** entity.

*EXPRESS* specification:

```
*)
ENTITY circle
  SUBTYPE OF (conic);
  radius    : positive_length_measure;
END_ENTITY;
(*
```

Attribute definitions:

**SELF\conic.position.location:** This inherited attribute defines the centre of the circle.

**radius:** The radius of the circle, which shall be greater than zero.

NOTE – See figure 3 for interpretation of attributes.

## 4.4.23 ellipse

An **ellipse** is a conic section defined by the lengths of the semi-major and semi-minor diameters and the position (center or mid point of the line joining the foci) and orientation of the curve.

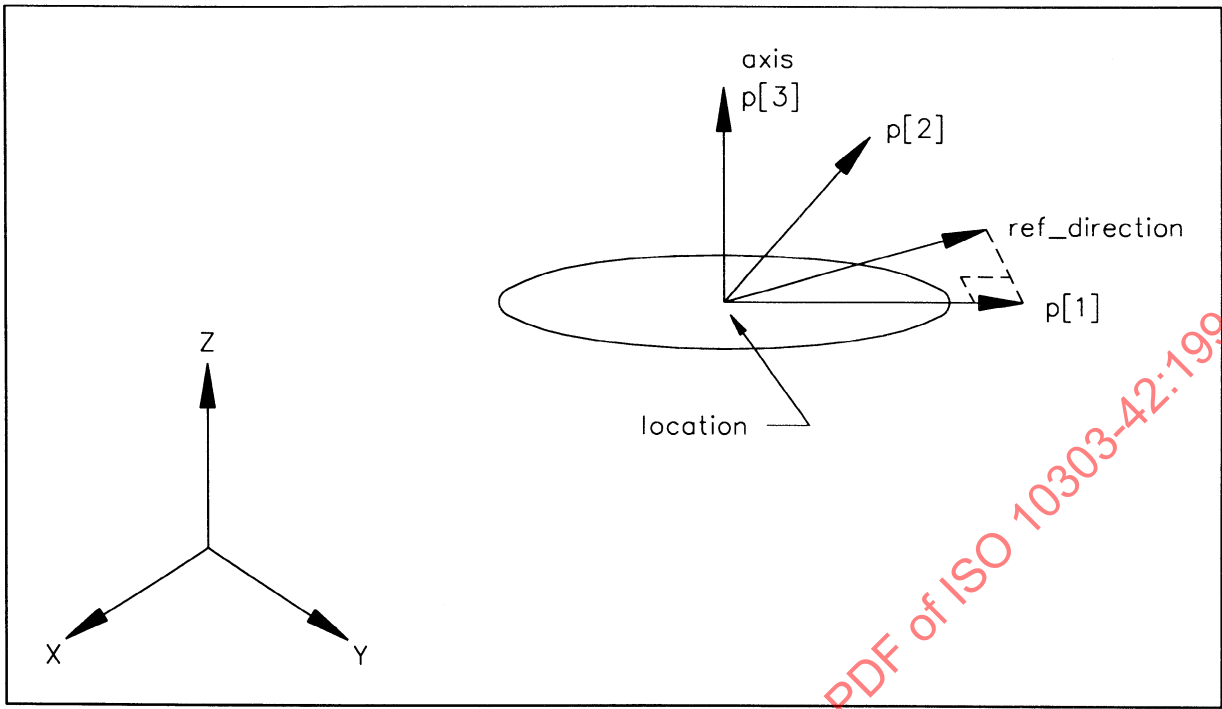


**Figure 4 – Ellipse**

Interpretation of the data shall be as follows:

$$\begin{aligned} \mathbf{C} &= \text{position.location} \\ \mathbf{x} &= \text{position.p[1]} \\ \mathbf{y} &= \text{position.p[2]} \\ \mathbf{z} &= \text{position.p[3]} \\ R_1 &= \text{semi\_axis\_1} \\ R_2 &= \text{semi\_axis\_2} \end{aligned}$$

and the ellipse is parametrised as

$$\boldsymbol{\lambda}(u) = \mathbf{C} + (R_1 \cos u)\mathbf{x} + (R_2 \sin u)\mathbf{y}$$

The parametrisation range is $0 \leq u \leq 360$ degrees.

In the placement coordinate system defined above the ellipse is the equation $\mathcal{C} = 0$, where

$$\mathcal{C}(x, y, z) = x^2/R_1^2 + y^2/R_2^2 - 1$$

The positive sense of the ellipse at any point is in the tangent direction, $\mathbf{T}$, to the curve at the point, where

$$\mathbf{T} = (-\mathcal{C}_y, \mathcal{C}_x, 0).$$

<u>*EXPRESS* specification</u>:

```
*)
ENTITY ellipse
```

```
  SUBTYPE OF (conic);
  semi_axis_1 : positive_length_measure;
  semi_axis_2 : positive_length_measure;
END_ENTITY;
(*
```

Attribute definitions:

**SELF\conic.position: conic.position.location** is the centre of the ellipse, and **conic.position.p[1]** the direction of the **semi_axis_1**.

**semi_axis_1:** The first radius of the ellipse which shall be positive.

**semi_axis_2:** The second radius of the ellipse which shall be positive.

NOTE – See figure 4 for interpretation of attributes.



### 4.4.24 hyperbola

A **hyperbola** is a conic section defined by the lengths of the major and minor radii and the position (mid-point of the line joining two foci) and orientation of the curve. Interpretation of the data shall be as follows:

$$\begin{aligned} \mathbf{C} &= \text{position.location} \\ \mathbf{x} &= \text{position.p[1]} \\ \mathbf{y} &= \text{position.p[2]} \\ \mathbf{z} &= \text{position.p[3]} \\ R_1 &= \text{semi\_axis} \\ R_2 &= \text{semi\_imag\_axis} \end{aligned}$$

and the hyperbola is parametrised as

$$\boldsymbol{\lambda}(u) = \mathbf{C} + (R_1 \cosh u)\mathbf{x} + (R_2 \sinh u)\mathbf{y}$$

The parametrisation range is $-\infty < u < \infty$.

In the placement coordinate system defined above, the hyperbola is represented by the equation $\mathcal{C} = 0$, where

$$\mathcal{C}(x, y, z) = x^2/R_1^2 - y^2/R_2^2 - 1$$

The positive sense of the hyperbola at any point is in the tangent direction, **T**, to the curve at the point, where

$$\mathbf{T} = (-\mathcal{C}_y, \mathcal{C}_x, 0).$$

The branch of the hyperbola represented is that pointed to by the **x** direction.

*EXPRESS* specification:

```
*)
ENTITY hyperbola
  SUBTYPE OF (conic);
  semi_axis      : positive_length_measure;
```

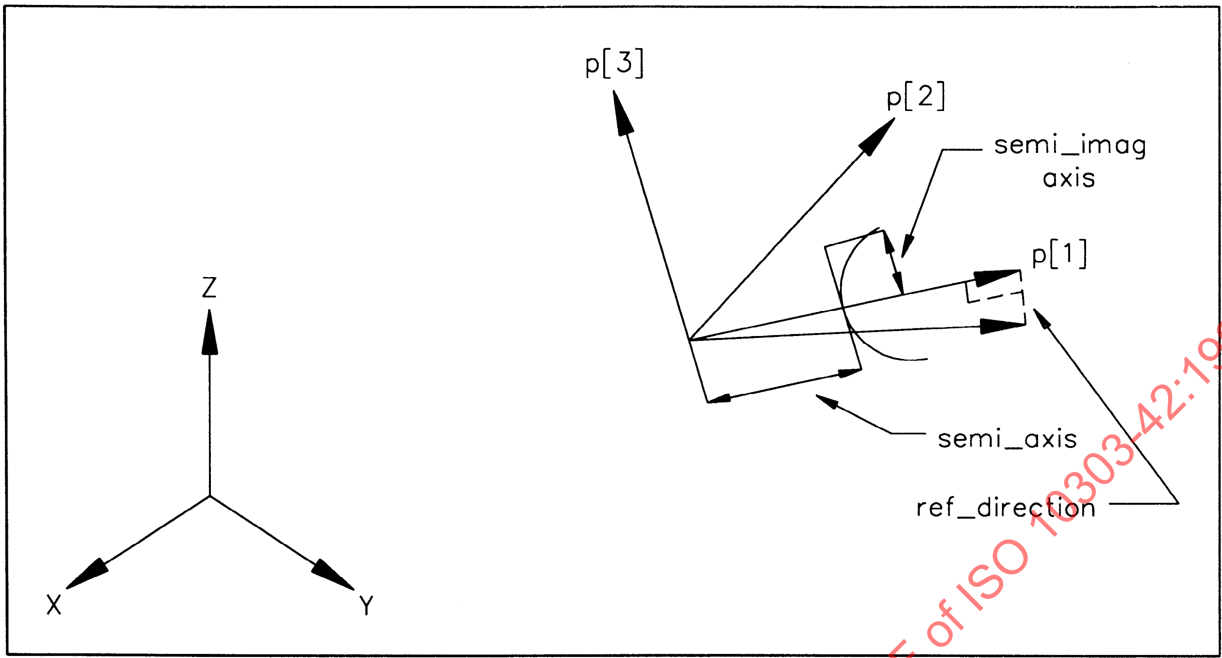


**Figure 5 – Hyperbola**



```
  semi_imag_axis : positive_length_measure;
END_ENTITY;
(*
```

Attribute definitions:

**SELF\conic.position:** The location and orientation of the curve.
**conic.position.location** is the centre of the hyperbola and **conic.position.p[1]** is in the direction of the semi-axis. The branch defined is on the side of **position.p[1]** positive.

**semi_axis:** The length of the semi axis of the hyperbola. This is positive and is half the minimum distance between the two branches of the hyperbola.

**semi_imag_axis:** The length of the semi imaginary axis of the hyperbola which shall be positive.

NOTE – See figure 5 for interpretation of attributes.

Formal propositions:

**WR1:** The length of the **semi_axis** shall be greater than zero.

**WR2:** The length of the **semi_imag_axis** shall be greater than zero.

### 4.4.25 parabola

A **parabola** is a conic section defined by its focal length, position (apex), and orientation. Interpretation of the data shall be as follows:

$$\begin{aligned} \mathbf{C} &= \text{position.location} \\ \mathbf{x} &= \text{position.p[1]} \\ \mathbf{y} &= \text{position.p[2]} \\ \mathbf{z} &= \text{position.p[3]} \\ F &= \text{focal\_dist} \end{aligned}$$

and the parabola is parametrised as

$$\boldsymbol{\lambda}(u) = \mathbf{C} + F(u^2\mathbf{x} + 2u\mathbf{y})$$

The parametrisation range is $-\infty < u < \infty$.

In the placement coordinate system defined above, the parabola is represented by the equation $\mathcal{C} = 0$,
where

$$\mathcal{C}(x, y, z) = 4Fx - y^2$$

The positive sense of the curve at any point is in the tangent direction, $\mathbf{T}$, to the curve at the point, where

$$\mathbf{T} = (-\mathcal{C}_y, \mathcal{C}_x, 0).$$

*EXPRESS* specification:

```
*)
ENTITY parabola
  SUBTYPE OF (conic);
  focal_dist : length_measure;
WHERE
  WR1: focal_dist <> 0.0;
END_ENTITY;
(*
```

Attribute definitions:

**SELF\conic.position:** The location and orientation of the curve. **conic.position.location** is the apex of the parabola and **conic.position.p[1]** is the axis of symmetry.

**focal_dist:** The distance of the focal point from the apex point.

NOTE – See figure 6 for interpretation of attributes.

Formal propositions:

**WR1:** The focal distance shall not be zero.

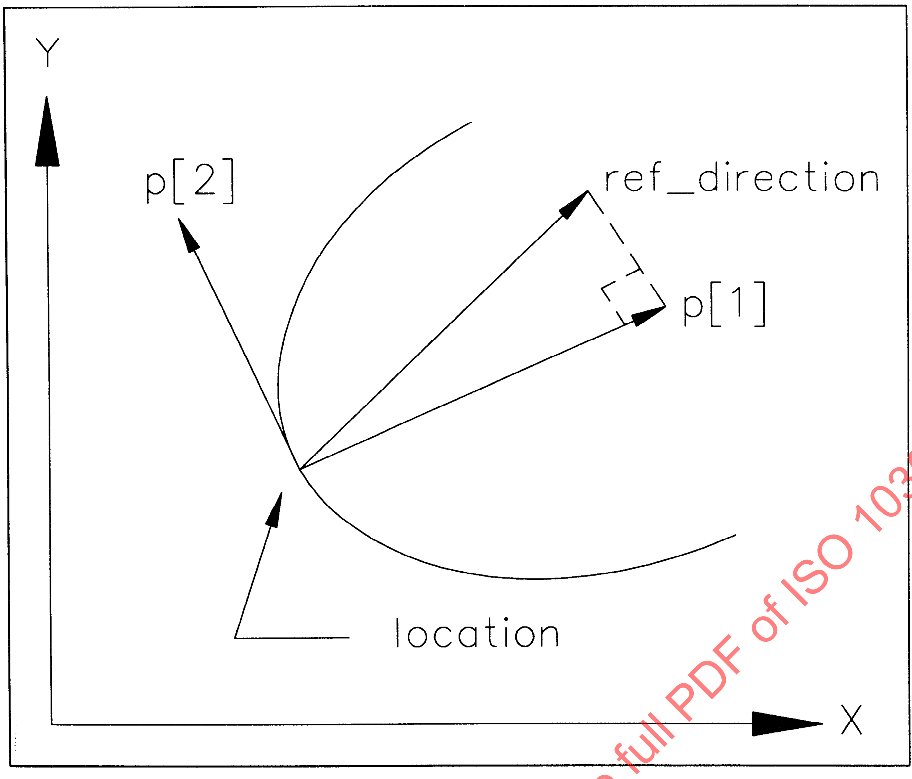


**Figure 6 – Parabola**



### 4.4.26 bounded_curve

A **bounded_curve** is a **curve** of finite arc length with identifiable end points.

*EXPRESS* specification:

```
*)
ENTITY bounded_curve
  SUPERTYPE OF (ONEOF(polyline, b_spline_curve, trimmed_curve,
                      bounded_pcurve, bounded_surface_curve, composite_curve))
  SUBTYPE OF (curve);
END_ENTITY;
(*
```

Informal propositions:

**IP1:** A bounded curve has finite arc length.

**IP2:** A bounded curve has a start point and an end point.

### 4.4.27 polyline

A **polyline** is a **bounded_curve** of $n-1$ linear segments, defined by a list of $n$ **points**, $P_1, P_2 \ldots P_n$.

The $i$th segment of the curve is parametrised as follows:

$$\boldsymbol{\lambda}(u) = \mathbf{P}_i(i-u) + \mathbf{P}_{i+1}(u+1-i), \qquad for 1 \leq i \leq n-1$$

where $i-1 \leq u \leq i$ and with parametric range of $0 \leq u \leq n-1$.

*EXPRESS* specification:

```
*)
ENTITY polyline
  SUBTYPE OF (bounded_curve);
  points : LIST [2:?] OF cartesian_point;
END_ENTITY;
(*
```

Attribute definitions:

**points:** The **points** defining the **polyline**.

### 4.4.28 b_spline_curve

A B-spline curve is a piecewise parametric polynomial or rational curve described in terms of control points and basis functions. The B-spline curve has been selected as the most stable format to represent all types of polynomial or rational parametric curves. With appropriate attribute values it is capable of representing single span or spline curves of explicit polynomial, rational, Bézier or B-spline type. The **b_spline_curve** has three special subtypes where the knots and knot multiplicities can be derived to provide simple default capabilities.

NOTES

1 – Identification of B-spline curve default values and subtypes is important for performance considerations and for efficiency issues in performing computations.

2 – A B-spline is *rational* if and only if the weights are not all identical; this can be represented by the **rational_b_spline_curve** subtype. If it is polynomial, the weights may be defaulted to all being 1.

3 – In the case where the B-spline curve is uniform, quasi-uniform or Bézier (including piecewise Bézier), the knots and knot multiplicities may be defaulted (i.e., non-existent in the data as specified by the attribute definitions).

4 – When the knots are defaulted, a difference of 1.0 between separate knots is assumed, and the effective parameter range for the resulting curve starts from 0.0. These defaults are provided by the subtypes.

5 – The knots and knot multiplicities shall not be defaulted in the non-uniform case.

6 – The defaulting of weights and knots are done independently of one another.

7 – Definitions of the B-spline basis functions $\mathbf{P}_i N_i^d(u)$ can be found in [E-1, E-2, E-3]. It should be noted that there is a difference in terminology between these references.

Interpretation of the data is as follows:

a) The curve, in the polynomial case, is given by:

$$\boldsymbol{\lambda}(u) = \sum_{i=0}^{k} \mathbf{P}_i N_i^d(u).$$

b) In the rational case all weights shall be positive and the curve is given by:

$$\boldsymbol{\lambda}(u) = \frac{\sum_{i=0}^{k} w_i \mathbf{P}_i N_i^d(u)}{\sum_{i=0}^{k} w_i N_i^d(u)}.$$

where

$k+1$ = number of control points,
$\mathbf{P}_i$ = control points,
$w_i$ = weights, and
$d$ = degree.

The knot array is an array of $(k+d+2)$ real numbers $[u_{-d}, ..., u_{k+1}]$, such that for all indices j in $[-d, k]$, $u_j <= u_{j+1}$. This array is obtained from the **knots_data** list by repeating each multiple knot according to the multiplicity. $N_i^d$, the $i$th normalised B-spline basis function of degree $d$, is defined on the subset
$[u_{i-d}, ..., u_{i+1}]$ of this array.

c) Let $L$ denote the number of distinct values amongst the $d+k+2$ knots in the knot list; $L$ will be referred to as the 'upper index on knots'. Let $m_j$ denote the multiplicity (i.e., number of repetitions) of the $j$th distinct knot. Then:

$$\sum_{i=1}^{L} m_i = d + k + 2.$$

All knot multiplicities except the first and the last shall be in the range $1 \ldots d$; the first and last may have a maximum value of $d+1$.

In evaluating the basis functions, a knot $u$ of, e.g., multiplicity 3 is interpreted as a sequence $u, u, u$, in the knot array.

The **b_spline_curve** has three special subtypes where the knots and knot multiplicities are derived to provide simple default capabilities.

NOTE – see figure 7 for further information on curve definition.

*EXPRESS* specification:

```
*)
ENTITY b_spline_curve
  SUPERTYPE OF (ONEOF(uniform_curve, b_spline_curve_with_knots,
```

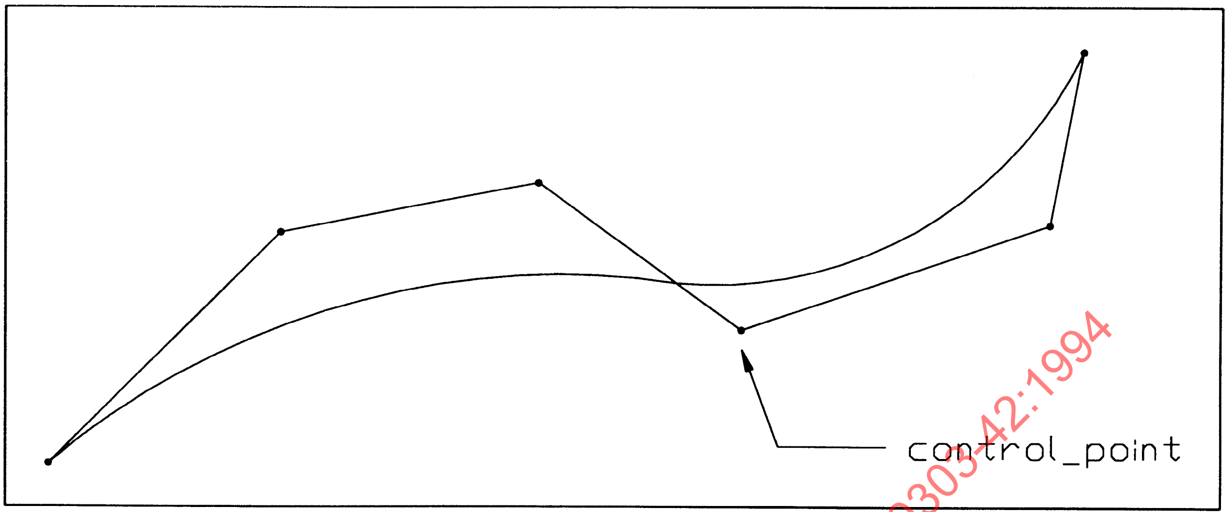


**Figure 7 – B-spline curve**



```
                          quasi_uniform_curve, bezier_curve)
                            ANDOR rational_b_spline_curve)
  SUBTYPE OF (bounded_curve);
  degree                    : INTEGER;
  control_points_list       : LIST [2:?] OF cartesian_point;
  curve_form                : b_spline_curve_form;
  closed_curve              : LOGICAL;
  self_intersect            : LOGICAL;
DERIVE
  upper_index_on_control_points : INTEGER
                              := (SIZEOF(control_points_list) - 1);
  control_points            : ARRAY [0:upper_index_on_control_points]
                                                   OF cartesian_point
                              := list_to_array(control_points_list,0,
                                           upper_index_on_control_points);
WHERE
  WR1: ('GEOMETRY_SCHEMA.UNIFORM_CURVE' IN TYPEOF(self)) OR
       ('GEOMETRY_SCHEMA.QUASI_UNIFORM_CURVE' IN TYPEOF(self)) OR
       ('GEOMETRY_SCHEMA.BEZIER_CURVE' IN TYPEOF(self)) OR
       ('GEOMETRY_SCHEMA.B_SPLINE_CURVE_WITH_KNOTS' IN TYPEOF(self));
END_ENTITY;
(*
```

Attribute definitions:

**degree:** The algebraic degree of the basis functions.

**control_points_list:** The list of control points for the curve.

**curve_form:** Used to identify particular types of curve; it is for information only. (See 4.3.4 for details).

**closed_curve:** Indication of whether the curve is closed; it is for information only.

**self_intersect:** Flag to indicate whether the curve self intersects or not; it is for information only.

**SELF\geometric_representation_item.dim:** The dimensionality of the coordinate space for the curve.

**upper_index_on_control_points:** The upper index on the array of control points; the lower index is 0. This value is derived from the list of control points.

**control_points:** The array of control points used to define the geometry of the curve. This is derived from the list of control points.

NOTE – Where part of the data is described as 'for information only' this implies that if there is any discrepancy between this information and the properties derived from the curve itself, the curve data takes precedence.

Formal propositions:

**WR1:** Any instantiation of this entity shall include one of the subtypes **b_spline_curve_with_knots**, **uniform_curve**, **quasi_uniform_curve** or **bezier_curve**.

## 4.4.29 b_spline_curve_with_knots

This is the subtype of **b_spline_curve** for which the knot values are explicitly given. This subtype shall be used to represent non-uniform B-spline curves and may be used for other knot types.

Let $L$ denote the number of distinct values amongst the $d+k+2$ knots in the knot list; $L$ will be referred to as the 'upper index on knots'. Let $m_j$ denote the multiplicity (i.e., number of repetitions) of the $j$th distinct knot. Then:

$$\sum_{i=1}^{L} m_i = d + k + 2.$$

All knot multiplicities except the first and the last shall be in the range $1 \dots d$; the first and last may have a maximum value of $d+1$.

In evaluating the basis functions, a knot $u$ of, e.g., multiplicity 3 is interpreted as a sequence $u, u, u,$ in the knot array.

*EXPRESS* specification:

```
*)
ENTITY b_spline_curve_with_knots
  SUBTYPE OF (b_spline_curve);
  knot_multiplicities  : LIST [2:?] OF INTEGER;
  knots                : LIST [2:?] OF parameter_value;
  knot_spec            : knot_type;
```

```
 DERIVE
   upper_index_on_knots : INTEGER := SIZEOF(knots);
 WHERE
   WR1: constraints_param_b_spline(degree, upper_index_on_knots,
                                   upper_index_on_control_points,
                                   knot_multiplicities, knots);
    WR2: SIZEOF(knot_multiplicities) = upper_index_on_knots;
END_ENTITY;
 (*
```

Attribute definitions:

**knot_multiplicities:** The multiplicities of the knots. This list defines the number of times each knot in the **knots** list is to be repeated in constructing the knot array.

**knots:** The list of distinct knots used to define the B-spline basis functions.

**knot_spec:** The description of the knot type. This is for information only.

**SELF\b_spline_curve.curve_form:** Used to identify particular types of curve; it is for information only. (See 4.3.4 for details).

**SELF\b_spline_curve.degree:** The algebraic degree of the basis functions.

**SELF\b_spline_curve.closed_curve:** Indication of whether the curve is closed; it is for information only.

**SELF\b_spline_curve.self_intersect:** Flag to indicate whether the curve self intersects or not; it is for information only.

**dim:** The dimensionality of the coordinate space for the curve.

**SELF\b_spline_curve.upper_index_on_control_points:** The upper index on the array of control points; the lower index is 0. This value is derived from the list of control points

**upper_index_on_knots:** The upper index on the knot arrays; the lower index is 1.

**SELF\b_spline_curve.control_points:** The array of control points used to define the geometry of the curve. This is derived from the list of control points.

NOTE – Where part of the data is described as 'for information only' this implies that if there is any discrepancy between this information and the properties derived from the curve itself, the curve data takes precedence.

Formal propositions:

**WR1: constraints_param_b_spline** returns TRUE if no inconsistencies in the parametrisation of the B-spline are found.

**WR2:** The number of elements in the knot multiplicities list shall be equal to the number of elements in the knots list.

### 4.4.30 uniform_curve

This is a special subtype of **b_spline_curve** in which the knots are evenly spaced. Suitable default values for the knots and knot multiplicities are derived in this case.

A B-spline is *uniform* if and only if all knots are of multiplicity 1 and they differ by a positive constant from the preceding knot. In this subtype the knot spacing is 1.0, starting at $-d$, where $d$ is the degree.

NOTE – If the B-spline curve is uniform and degree=1, the B-spline is equivalent to a **polyline**.

*EXPRESS* specification:

```
*)
ENTITY uniform_curve
  SUBTYPE OF (b_spline_curve);
END_ENTITY;
(*
```

NOTE – The value k_up may be required for the upper index on the knot and knot multiplicity lists. This is computed from the degree and the number of control points.
k_up = SELF\b_spline_curve.upper_index_on_control_points + degree + 2.
If required, the knots and knot multiplicities can be computed by the function calls:
default_b_spline_knots(SELF\b_spline_curve.degree, k_up,uniform_knots), and
default_b_spline_knot_mult(SELF\b_spline_curve.degree,k_up, uniform_knots).

### 4.4.31 quasi_uniform_curve

This is a special subtype of **b_spline_curve** in which the knots are evenly spaced, and except for the first and last, have multiplicity 1. Suitable default values for the knots and knot multiplicities are derived in this case.

A B-spline is *quasi-uniform* if and only if the knots are of multiplicity (degree+1) at the ends, of multiplicity 1 elsewhere, and they differ by a positive constant from the preceding knot. A quasi-uniform B-spline curve which has only two knots represents a Bézier curve. In this subtype the knot spacing is 1.0, starting at 0.0.

*EXPRESS* specification:

```
*)
ENTITY quasi_uniform_curve
  SUBTYPE OF (b_spline_curve);
END_ENTITY;
(*
```

NOTE – The value k_up may be required for the upper index on the knot and knot multiplicity lists. This is computed from the degree and the number of control points.

k_up = SELF\b_spline_curve.upper_index_on_control_points - degree + 2.
If required, the knots and knot multiplicities can then be computed by the function calls:

default_b_spline_knots(SELF\b_spline_curve.degree,k_up, quasi_uniform_knots), and
default_b_spline_knot_mult(SELF\b_spline_curve.degree,k_up, quasi_uniform_knots).

### 4.4.32 bezier_curve

This subtype represents in the most general case a piecewise Bézier curve. This is a special type of curve which can be represented as a subtype of **b_spline_curve** in which the knots are evenly spaced and have high multiplicities. Suitable default values for the knots and knot multiplicities are derived in this case.

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 is 1.0, starting at 0.0. A piecewise Bézier curve which has only two knots, 0.0 and 1.0, each of multiplicity (degree+1), is a simple Bézier curve.

NOTES

1 – A simple Bézier curve can be defined as a B-spline curve with knots by the following data:

| | |
|---|---|
| degree | $(d)$ |
| upper index on control points | (equal to $d$) |
| control points | ($d+1$ cartesian points) |
| knot type | (equal to quasi uniform knots) |
| knot multiplicities | $(d+1, d+1)$ |
| knots | (0.0, 1.0) |

No other data are needed, except for a rational Bézier curve. In this case the weights data (($d+1$) REALs) shall be given.

2 – It should be noted that every piecewise Bézier curve has an equivalent representation as a B-spline curve. Because of problems with non-uniform knots not every B-spline curve can be represented as a piecewise Bézier curve.

To define a piecewise Bézier curve as a B-spline:

– The first knot is 0.0 with multiplicity ($d+1$).

– The next knot is 1.0 with multiplicity $d$ (we have now defined the knots for one segment, unless it is the last one).

– The next knot is 2.0 with multiplicity $d$ (we have now defined the knots for two segments, again unless the second is the last one).

– Continue to the end of the last segment, call it the $n$-th segment, at the end of which a knot with value $n$, multiplicity ($d+1$) is added.

EXAMPLES

6 – A one-segment cubic Bézier curve would have knot sequence (0,1) with multiplicity sequence (4,4).

7 – A two-segment cubic piecewise Bézier curve would have knot sequence (0,1,2) with multiplicity sequence (4,3,4).

3 – For the piecewise Bézier case, if $d$ is the degree, $k+1$ is the number of control points, $m$ is the number of knots with multiplicity $d$, and $N$ is the total number of knots for the spline, then

$$\begin{aligned}(d+2+k) &= N \\ &= (d+1)+md+(d+1) \\ thus, m &= (k-d)/d\end{aligned}$$

Thus, the knot sequence is $(0, 1, \ldots m, (m+1))$ with multiplicities $(d+1, d, \ldots d, d+1)$.

*EXPRESS* specification:

```
*)
ENTITY bezier_curve
  SUBTYPE OF (b_spline_curve);
END_ENTITY;
(*
```

NOTE 4 – The value k_up may be required for the upper index on the knot and knot multiplicity lists. This is computed from the degree and the number of control points.

$$k\_up = \frac{SELF\backslash b\_spline\_curve.upper\_index\_on\_control\_points}{SELF\backslash b\_spline\_curve.degree} + 1.$$

If required, the knots and knot multiplicities can then be computed by the function calls: default_b_spline_knots(SELF\b_spline_curve.degree,k_up, piecewise_bezier_knots), and default_b_spline_knot_mult(SELF\b_spline_curve.degree,k_up, piecewise_bezier_knots).

## 4.4.33 rational_b_spline_curve

A **rational_b_spline_curve** is a piecewise parametric rational curve described in terms of control points and basis functions. This subtype is instantiated with one of the other subtypes of **b_spline_curve** which explicitly or implicitly provide the knot values used to define the basis functions.

All weights shall be positive and the curve is given by:

$$\boldsymbol{\lambda}(u) = \frac{\sum_{i=0}^{k} w_i \mathbf{P}_i N_i^d(u)}{\sum_{i=0}^{k} w_i N_i^d(u)}.$$

where

$$\begin{aligned} k+1 &= \text{number of control points,} \\ \mathbf{P}_i &= \text{control points,} \\ w_i &= \text{weights, and} \\ d &= \text{degree.} \end{aligned}$$

*EXPRESS* specification:

```
*)
ENTITY rational_b_spline_curve
  SUBTYPE OF (b_spline_curve);
  weights_data : LIST [2:?] OF REAL;

DERIVE
  weights                : ARRAY [0:upper_index_on_control_points] OF REAL
                                    := list_to_array(weights_data,0,
                                           upper_index_on_control_points);
WHERE
  WR1:  SIZEOF(weights_data) = SIZEOF(SELF\b_spline_curve.control_points_list);
  WR2:  curve_weights_positive(SELF);
END_ENTITY;
(*
```

Attribute definitions:

**weights_data:** The supplied values of the weights. See the derived attribute **weights**.

**SELF\b_spline_curve.degree:** The algebraic degree of the basis functions.

**SELF\b_spline_curve.curve_form:** Used to identify particular types of curve; it is for information only. (See 4.3.4 for details.)

**SELF\b_spline_curve.closed_curve:** Indication of whether the curve is closed; it is for information only.

**SELF\b_spline_curve.self_intersect:** Flag to indicate whether the curve self intersects or not; it is for information only.

**SELF\b_spline_curve.upper_index_on_control_points:** The upper index on the array of control points; the lower index is 0. This value is derived from the list of control points

**SELF\b_spline_curve.control_points:** The array of control points used to define the geometry of the curve.This is derived from the list of control points

**weights:** The array of weights associated with the control points. This is derived from the **weights_data**

NOTE – Where part of the data is described as 'for information only' this implies that if there is any discrepancy between this information and the properties derived from the curve itself the curve data takes precedence.

Formal propositions:

**WR1:** There shall be the same number of weights as control points.

**WR2:** All the weights shall have values greater than 0.0.

### 4.4.34 trimmed_curve

A trimmed curve is a bounded curve which is created by taking a selected portion, between two identified points, of the associated basis curve. The basis curve itself is unaltered and more than one trimmed curve may reference the same basis curve. Trimming points for the curve may be identified:

- by parametric value;
- by geometric position;
- by both of the above.

At least one of these shall be specified at each end of the curve. The **sense** makes it possible to unambiguously define any segment of a closed curve such as a circle. The combinations of sense and ordered end points make it possible to define four distinct directed segments connecting two different points on a circle or other closed curve. For this purpose cyclic properties of the parameter range are assumed; for example, 370 degrees is equivalent to 10 degrees.

The trimmed curve has a parametrisation which is inherited from that of the particular basis curve referenced. More precisely the parameter $s$ of the trimmed curve is derived from the parameter $t$ of the basis curve as follows:

If sense is TRUE: $s = t - t_1$.
If sense is FALSE: $s = t_2 - t$.

In the above equations $t_1$ is the value given by trim_1 or the parameter value corresponding to point_1 and $t_2$ is the parameter value given by trim_2 or the parameter corresponding to point_2. The resultant trimmed curve has a parameter $s$ ranging from 0 at the first trimming point to $|t_2 - t_1|$ at the second trimming point.

NOTES

1 – In the case of a closed basis curve, it may be necessary to increment $t_1$ or $t_2$ by the parametric length for consistency with the sense flag.

2 – For example:
(a) If **sense_agreement** = TRUE and $t_2 < t_1$, $t_2$ should be increased by the parametric length.
(b) If **sense_agreement** = FALSE and $t_1 < t_2$, $t_1$ should be increased by the parametric length.

*EXPRESS specification:*

```
*)
ENTITY trimmed_curve
  SUBTYPE OF (bounded_curve);
  basis_curve           : curve;
  trim_1                : SET[1:2] OF trimming_select;
  trim_2                : SET[1:2] OF trimming_select;
  sense_agreement       : BOOLEAN;
  master_representation : trimming_preference;
WHERE
```

```
   WR1: (HIINDEX(trim_1) = 1) XOR (TYPEOF(trim_1[1]) <> TYPEOF(trim_1[2]));
   WR2: (HIINDEX(trim_2) = 1) XOR (TYPEOF(trim_2[1]) <> TYPEOF(trim_2[2]));
 END_ENTITY;
 (*
```

Attribute definitions:

**basis_curve:** The **curve** to be trimmed. For curves with multiple representations any parameter values given as **trim_1** or **trim_2** refer to the master representation of the **basis_curve** only.

**trim_1:** The first trimming point which may be specified as a cartesian point (point_1), as a real parameter value (parameter_1 = $t_1$), or both.

**trim_2:** The second trimming point which may be specified as a cartesian point (point_2), as a real parameter value (parameter_2 = $t_2$), or both.

**sense_agreement:** Flag to indicate whether the direction of the trimmed curve agrees with or is opposed to the direction of **basis_curve**.

– sense agreement = TRUE if the curve is being traversed in the direction of increasing parametric value;

– sense agreement = FALSE otherwise. For an open curve, sense agreement = FALSE if $t_1 > t_2$. If $t_2 > t_1$, sense agreement = TRUE. The sense information is redundant in this case but is essential for a closed curve.

**master_representation:** Where both parameter and point are present at either end of the curve this indicates the preferred form. Multiple representations provide the ability to communicate data in more than one form, even though the data are expected to be geometrically identical. (See 4.3.8.)

NOTE – The master_representation attribute acknowledges the impracticality of ensuring that multiple forms are indeed identical and allows the indication of a preferred form. This would probably be determined by the creator of the data. All characteristics, such as parametrisation, domain, and results of evaluation, for an entity having multiple representations, are derived from the master representation. Any use of the other representations is a compromise for practical considerations.

Formal propositions:

**WR1:** Either a single value is specified for **trim_1**, or, the two trimming values are of different types (point and parameter).

**WR2:** Either a single value is specified for **trim_2**, or, the two trimming values are of different types (point and parameter).

Informal propositions:

**IP1:** Where both the parameter value and the cartesian point exist for **trim_1** or **trim_2** they shall be consistent, i.e., the **basis_curve** evaluated at the parameter value shall coincide with the specified point.

**IP2:** When a cartesian point is specified by **trim_1** or by **trim_2**, it shall lie on the **basis_curve**.

**IP3:** Except in the case of a closed **basis_curve**, where both parameter_1 and parameter_2 exist, they shall be consistent with the sense flag, i.e., sense = (parameter_1 < parameter_2).

**IP4:** If both parameter_1 and parameter_2 exist, parameter_1 <> parameter_2.

**IP5:** When a parameter value is specified by **trim_1** or **trim_2**, it shall lie within the parametric range of the **basis_curve**.

### 4.4.35 composite_curve

A **composite_curve** is a collection of curves joined end-to-end. The individual segments of the curve are themselves defined as **composite_curve_segments**. The parametrisation of the composite curve is an accumulation of the parametric ranges of the referenced bounded curves. The first segment is parametrised from 0 to $l_1$, and, for $i \geq 2$, the $i^{th}$ segment is parametrised from

$$\sum_{k=1}^{k=i-1} l_k \quad to \quad \sum_{k=1}^{k=i} l_k,$$

where $l_k$ is the parametric length (i.e., difference between maximum and minimum parameter values) of the curve underlying the $k^{th}$ segment. Let $T$ denote the parameter for the **composite_curve**. Then, if the $i$th segment is not a **reparametrised_composite_curve_segment**, $T$ is related to the parameter $t_i$, $t_{i0} \leq t_i \leq t_{i1}$, for the $i$th segment by the equations:

$$T = \sum_{k=1}^{k=i-1} l_k + t_i - t_{i0},$$

if **segments[i].same_sense** = TRUE;

$$T = \sum_{k=1}^{k=i-1} l_k + t_{i1} - t_i,$$

if **segments[i].same_sense** = FALSE.

If **segments[i]** is of type **reparametrised_composite_curve_segment**,

$$T = \sum_{k=1}^{k=i-1} l_k + \tau,$$

Where $\tau$ is defined in 4.4.37.

*EXPRESS* specification:

```
*)
ENTITY composite_curve
  SUBTYPE OF (bounded_curve);
  segments       : LIST [1:?] OF composite_curve_segment;
  self_intersect : LOGICAL;
DERIVE
  n_segments     : INTEGER := SIZEOF(segments);
  closed_curve   : LOGICAL
                 := segments[n_segments].transition <> discontinuous;
WHERE
  WR1: ((NOT closed_curve) AND (SIZEOF(QUERY(temp <* segments |
              temp.transition = discontinuous)) = 1)) OR
           ((closed_curve) AND (SIZEOF(QUERY(temp <* segments |
              temp.transition = discontinuous)) = 0));
END_ENTITY;
(*
```

Attribute definitions:

**n_segments:** The number of component curves.

**segments:** The component bounded curves, their transitions and senses. The transition attribute for the last segment defines the transition between the end of the last segment and the start of the first; this transition attribute may take the value **discontinuous**, which indicates an open curve. (See 4.3.2).

**self_intersect:** Indication of whether the curve intersects itself or not; this is for information only.

**dim:** The dimensionality of the coordinate space for the composite curve. This is an inherited attribute from the geometric representation item supertype.

**closed_curve:** Indication of whether the curve is closed or not; this is derived from the transition code on the last segment.

NOTE – See figure 8 for further information on attributes.

Formal propositions:

**WR1:** No transition code shall be discontinuous, except for the last code of an open curve.

Informal propositions:

**IP1:** The **same_sense** attribute of each segment correctly specifies the senses of the component curves. When traversed in the direction indicated by **same_sense**, the segments shall join end-to-end.

## 4.4.36 composite_curve_segment

A **composite_curve_segment** is a bounded curve together with transition information which is used to construct a **composite_curve**.

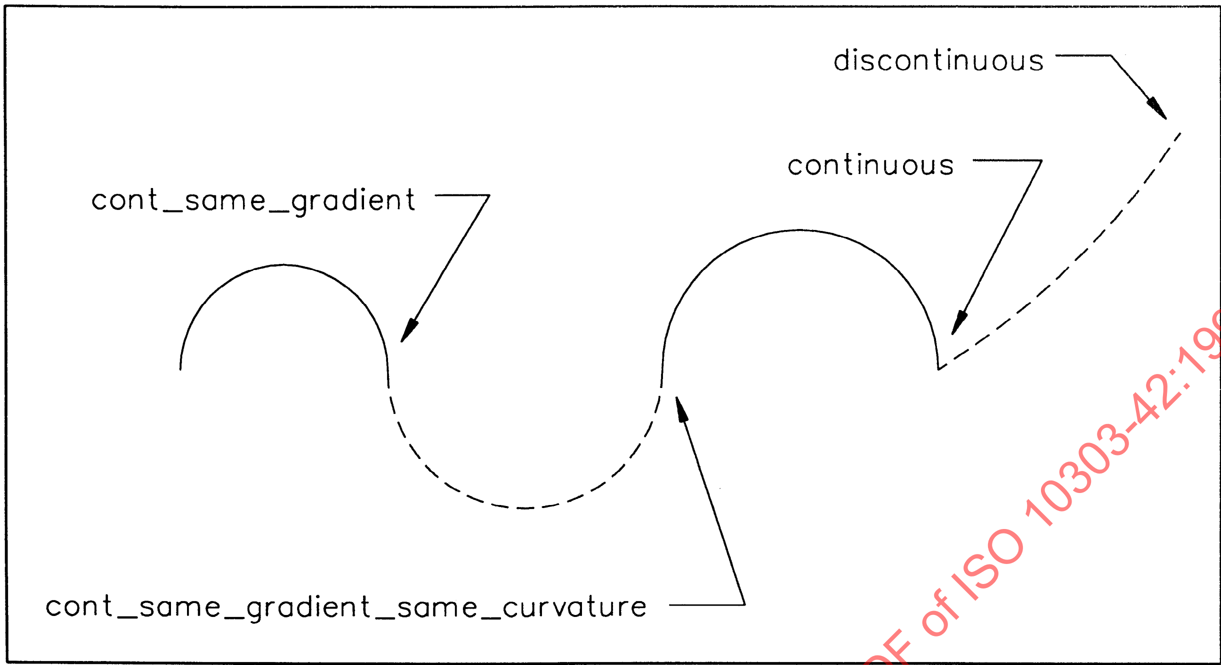


**Figure 8 – Composite curve**



*EXPRESS* specification:

```
*)
ENTITY composite_curve_segment;
  transition     : transition_code;
  same_sense     : BOOLEAN;
  parent_curve   : curve;
INVERSE
  using_curves   : BAG[1:?] OF composite_curve FOR segments;
WHERE
  WR1 : ('GEOMETRY_SCHEMA.BOUNDED_CURVE' IN TYPEOF(parent_curve));
END_ENTITY;
(*
```

Attribute definitions:

**transition:** The state of transition (i.e., geometric continuity from the last point of this segment to the first point of the next segment) in a composite curve.

**same_sense:** An indicator of whether or not the sense of the segment agrees with, or opposes, that of the parent curve. If **same_sense** is false, the point with highest parameter value is taken as the first point of the segment.

**parent_curve:** The bounded curve which defines the geometry of the segment.

**using_curves:** The set of **composite_curve**s which use this **composite_curve_segment** as a segment. This set shall not be empty.

Formal propositions:

**WR1:** The **parent_curve** shall be a **bounded_curve**.

## 4.4.37 reparametrised_composite_curve_segment

The **reparametrised_composite_curve_segment** is a special subtype of **composite_curve_segment** which provides the capability to re-define its parametric length without changing its geometry. Let $l$ = **param_length**.

If $t_0 \leq t \leq t_1$ is the parameter range of **parent_curve**, the new parameter $\tau$ for the **reparametrised_compos** is given by the equations:

$$\tau = \frac{t - t_0}{t_1 - t_0} l,$$

if **same_sense** = TRUE;

$$\tau = \frac{t_1 - t}{t_1 - t_0} l,$$

if **same_sense** = FALSE.

*EXPRESS* specification:

```
*)
ENTITY reparametrised_composite_curve_segment
  SUBTYPE OF (composite_curve_segment);
  param_length : parameter_value;
WHERE
  WR1: param_length > 0.0;
END_ENTITY;
(*
```

Attribute definitions:

**param_length:** The new parametric length of the segment. The segment is given a simple linear reparametrisation from 0.0 at the first point to **param_length** at the last point. The parametrisation of the composite curve constructed using this segment is defined in terms of **param_length**.

Formal propositions:

**WR1:** The **param_length** shall be greater than zero.

## 4.4.38 pcurve

A **pcurve** is a 3D curve defined by means of a 2D curve in the parameter space of a surface. If the curve is parametrised by the function $(u, v) = f(t)$, and the surface is parametrised by the function $(x, y, z) = g(u, v)$, the **pcurve** is parametrised by the function $(x, y, z) = g(f(t))$.

A **pcurve** definition contains a reference to its **basis_surface** and an indirect reference to a 2D curve through a **definitional_representation** entity. The 2D curve, being in parameter space, is not in the context of the basis surface. Thus a direct reference is not possible. For the 2D curve the variables involved are $u$ and $v$, which occur in the parametric representation of the **basis_surface** rather than $x, y$ Cartesian coordinates. The curve is only defined within the parametric range of the surface.

*EXPRESS* specification:

```
*)
ENTITY pcurve
  SUBTYPE OF (curve);
  basis_surface      : surface;
  reference_to_curve : definitional_representation;
WHERE
  WR1: SIZEOF(reference_to_curve\representation.items) = 1;
  WR2: 'GEOMETRY_SCHEMA.CURVE' IN TYPEOF
                 (reference_to_curve\representation.items[1]);
  WR3: reference_to_curve\representation.
                items[1]\geometric_representation_item.dim =2;
END_ENTITY;
(*
```

Attribute definitions:

**basis_surface:** The surface in whose parameter space the curve is defined.

**reference_to_curve:** The reference to the parameter space curve which defines the **pcurve**.

Formal propositions:

**WR1:** The set of items in the **definitional_representation** entity corresponding to the **reference_to_curve** shall have exactly one element.

**WR2:** The unique item in the set shall be a curve.

**WR3:** The dimensionality of this parameter space curve shall be 2.

## 4.4.39 bounded_pcurve

A **bounded_pcurve** is special subtype of **pcurve** which also has the properties of a **bounded_curve**.

*EXPRESS* specification:

```
*)
ENTITY bounded_pcurve
  SUBTYPE OF (pcurve, bounded_curve);
 WHERE
   WR1: ('GEOMETRY_SCHEMA.BOUNDED_CURVE' IN
                TYPEOF(SELF\pcurve.reference_to_curve.items[1]));
 END_ENTITY;
(*
```

Formal propositions:

**WR1:** The referenced curve of the **pcurve** supertype shall be of type **bounded_curve**. This ensures that the **bounded_pcurve** is of finite arc length.

### 4.4.40 surface_curve

A **surface_curve** is a curve on a surface. The curve is represented as a curve (**curve_3d**) in three-dimensional space and possibly as a curve, corresponding to a pcurve, in the two-dimensional parametric space of a surface. The ability of this curve to reference a list of 1 or 2 **pcurve_or_surface**s enables this entity to define either a curve on a single surface, or an intersection curve which has two distinct surface associations. A 'seam' on a closed surface can also be represented by this entity; in this case each **associated_geometry** will be a pcurve lying on the same surface. Each pcurve, if it exists, shall be parametrised to have the same sense as **curve_3d**. The surface curve takes its parametrisation directly from either **curve_3d** or a **pcurve** as indicated by the attribute master representation.

NOTE – Because of the ANDOR relationship with the **bounded_surface_curve** subtype an instance of a **surface_curve** may be any one of the following:

- a **surface_curve**;
- a bounded_surface_curve;
- an **intersection_curve**;
- an **intersection_curve** and **bounded_surface_curve**;
- a **seam_curve**;
- a **seam_curve** and **bounded_surface_curve**.

*EXPRESS* specification:

```
*)
ENTITY surface_curve
  SUPERTYPE OF (ONEOF(intersection_curve, seam_curve) ANDOR
                                         bounded_surface_curve)
  SUBTYPE OF (curve);
  curve_3d               : curve;
  associated_geometry    : LIST[1:2] OF pcurve_or_surface;
  master_representation  : preferred_surface_curve_representation;
DERIVE
  basis_surface          : SET[1:2] OF surface
                         := get_basis_surface(SELF);
WHERE
  WR1: curve_3d.dim = 3;
  WR2: ('GEOMETRY_SCHEMA.PCURVE' IN TYPEOF(associated_geometry[1])) OR
                     (master_representation <> pcurve_s1);
  WR3: ('GEOMETRY_SCHEMA.PCURVE' IN TYPEOF(associated_geometry[2])) OR
                     (master_representation <> pcurve_s2);
  WR4: NOT ('GEOMETRY_SCHEMA.PCURVE' IN TYPEOF(curve_3d));
END_ENTITY;
(*
```

Attribute definitions:

**curve_3d:** The curve which is the three-dimensional representation of the **surface_curve**.

**associated_geometry:** A list of one or two pcurves or surfaces which define the surface or surfaces associated with the surface curve. Two elements in this list indicate that the curve has two surface associations which need not be two distinct surfaces. When a pcurve is selected, it identifies a surface and also associates a basis curve in the parameter space of this surface.

**master_representation:** Indication of representation "preferred". The master representation defines the curve used to determine the unique parametrisation of the surface curve.
The master representation takes one of the values curve_3d, pcurve_s1 or pcurve_s2 to indicate a preference for the 3D curve, or the first or second pcurve, in the associated geometry list, respectively. Multiple representations provide the ability to communicate data in more than one form, even though the data is expected to be geometrically identical.

NOTE – The master_representation attribute acknowledges the impracticality of ensuring that multiple forms are indeed identical and allows the indication of a preferred form. This would probably be determined by the creator of the data. All characteristics, such as parametrisation, domain, and results of evaluation, for an entity having multiple representations, are derived from the master representation. Any use of the other representations is a compromise for practical considerations.

**basis_surface:** The surface, or surfaces on which the **surface_curve** lies. This is determined from the **associated_geometry** list.

Formal propositions:

**WR1: curve_3d** shall be defined in three-dimensional space.

**WR2: pcurve_s1** shall only be nominated as the master representation if the first element of the associated geometry list is a pcurve.

**WR3: pcurve_s2** shall only be nominated as the master representation if the second element of the associated geometry list is a pcurve. This also requires that **pcurve_s2** shall not be nominated when the associated geometry list contains a single element.

**WR4: curve_3d** shall not be a **pcurve**.

Informal propositions:

**IP1:** Where **curve_3d** and one or more **pcurve**s exist they shall represent the same mathematical point set. (i.e., They shall coincide geometrically but may differ in parametrisation.)

**IP2: curve_3d** and any associated pcurves shall agree with respect to their senses.

### 4.4.41 intersection_curve

An **intersection_curve** is a curve which results from the intersection of two surfaces. It is represented as a special subtype of the **surface_curve** entity having two distinct surface associations defined via the associated geometry list.

*EXPRESS specification*:

```
*)
ENTITY intersection_curve
  SUBTYPE OF (surface_curve);
WHERE
  WR1: SIZEOF(SELF\surface_curve.associated_geometry) = 2;
  WR2: associated_surface(SELF\surface_curve.associated_geometry[1]) <>
            associated_surface(SELF\surface_curve.associated_geometry[2]);
END_ENTITY;
(*
```

Formal propositions:

**WR1:** The intersection curve shall have precisely two associated geometry elements.

**WR2:** The two associated geometry elements shall be related to distinct surfaces. These are the surfaces which define the intersection curve.

### 4.4.42 seam_curve

A **seam_curve** is a curve on a closed parametric surface which has two distinct representations as constant parameter curves at the two extremes of the parameter range for the surface. For example the 'seam' on a cylinder has representations as the lines u = 0 or u = 360 degrees in parameter space.

*EXPRESS specification*:

```
*)
ENTITY seam_curve
  SUBTYPE OF (surface_curve);
WHERE
  WR1: SIZEOF(SELF\surface_curve.associated_geometry) = 2;
  WR2: associated_surface(SELF\surface_curve.associated_geometry[1]) =
         associated_surface(SELF\surface_curve.associated_geometry[2]);
  WR3: 'GEOMETRY_SCHEMA.PCURVE' IN
         TYPEOF(SELF\surface_curve.associated_geometry[1]);
  WR4: 'GEOMETRY_SCHEMA.PCURVE' IN
         TYPEOF(SELF\surface_curve.associated_geometry[2]);
END_ENTITY;
(*
```

Formal propositions:

**WR1:** The seam curve shall have precisely two **associated_geometry**s.

**WR2:** The two **associated_geometry**s shall be related to the same surface.

**WR3:** The first **associated_geometry** shall be a **pcurve**.

**WR4:** The second **associated_geometry** shall be a **pcurve**.

### 4.4.43 bounded_surface_curve

A **bounded_surface_curve** is a specialised subtype of **surface_curve** which also has the properties of a **bounded_curve**.

*EXPRESS* specification:

```
*)
ENTITY bounded_surface_curve
  SUBTYPE OF (surface_curve, bounded_curve);
WHERE
  WR1: ('GEOMETRY_SCHEMA.BOUNDED_CURVE' IN
            TYPEOF(SELF\surface_curve.curve_3d));
END_ENTITY;
(*
```

Formal propositions:

**WR1:** The **curve_3d** attribute of the **surface_curve** supertype shall be a **bounded_curve**.



### 4.4.44 composite_curve_on_surface

A **composite_curve_on_surface** is a collection of segments which are curves on a surface. Each segment shall lie on the basis surface, and may be

- a **surface_curve** or
- a **pcurve** or
- a **composite_curve_on_surface**.

NOTE – A **composite_curve_on_surface** can be included as the **parent_curve** attribute of a **composite_curve_segment** since it is a bounded curve subtype.

There shall be at least positional continuity between adjacent segments. The parametrisation of the composite curve is obtained from the accumulation of the parametric ranges of the segments. The first segment is parametrised from 0 to $l_1$, and, for $i \geq 2$, the $i^{th}$ segment is parametrised from

$$\sum_{k=1}^{k=i-1} l_k \quad \text{to} \quad \sum_{k=1}^{k=i} l_k,$$

where $l_k$ is the parametric length (i.e., difference between maximum and minimum parameter values) of the $k^{th}$ curve segment.

*EXPRESS* specification:

```
*)
ENTITY composite_curve_on_surface
  SUPERTYPE OF(boundary_curve)
  SUBTYPE OF (composite_curve);

DERIVE
  basis_surface : SET[0:2] OF surface :=
             get_basis_surface(SELF);
```

```
 WHERE
   WR1: SIZEOF(basis_surface) > 0;
   WR2: constraints_composite_curve_on_surface(SELF);
 END_ENTITY;
(*
```

Attribute definitions:

**basis_surface:** The surface on which the composite curve is defined.

**SELF\composite_curve.n_segments:** The number of component curves.

**SELF\composite_curve.segments:** The component bounded curves, their transitions and senses. The transition for the last segment defines the transition between the end of the last segment and the start of the first; this element may take the value **discontinuous**, which indicates an open curve. (See 4.3.2.)

**SELF\composite_curve.self_intersect:** Indication of whether the curve intersects itself or not.

**SELF\composite_curve.dim:** The dimensionality of the coordinate space for the composite curve.

**SELF\composite_curve.closed_curve:** Indication of whether the curve is closed or not.

Formal propositions:

**WR1:** The **basis_surface** SET shall contain at least one surface. This ensures that all segments reference curves on the same surface.

**WR2:** Each segment shall reference a **pcurve**, or a **surface_curve**, or a **composite_curve_on_surface**.

Informal propositions:

**IP1:** Each **parent_curve** referenced by a **composite_curve_on_surface** segment shall be a curve on surface and a bounded curve.

### 4.4.45 offset_curve_2d

An **offset_curve_2d** is a curve at a constant distance from a basis curve in two-dimensional space. This entity defines a simple plane-offset curve by offsetting by **distance** along the normal to **basis_curve** in the plane of **basis_curve**.

The underlying curve shall have a well-defined tangent direction at every point. In the case of a composite curve, the transition code between each segment shall be **cont_same_gradient** or **cont_same_gradient_same_curvature**.

NOTE – The **offset_curve_2d** may differ in nature from the **basis_curve**; the offset of a non self-intersecting curve can be self intersecting. Care should be taken to ensure that the offset to a continuous curve does not become discontinuous.

The **offset_curve_2d** takes its parametrisation from the **basis_curve**. The **offset_curve_2d** is parametrised as

$$\boldsymbol{\lambda}(u) = \mathbf{C}(u) + d(\mathbf{orthogonal\_complement}(\mathbf{T})),$$

where **T** is the unit tangent vector to the basis curve $\mathbf{C}(u)$ at parameter value $u$, and $d$ is **distance**. The underlying curve shall be two-dimensional.

*EXPRESS* specification:

```
*)
ENTITY offset_curve_2d
  SUBTYPE OF (curve);
  basis_curve    : curve;
  distance       : length_measure;
  self_intersect : LOGICAL;
WHERE
  WR1: basis_curve.dim = 2;
END_ENTITY;
(*
```

Attribute definitions:

**basis_curve:** The curve that is being offset.

**distance:** The distance of the offset curve from the basis curve. **distance** may be positive, negative or zero. A positive value of **distance** defines an offset in the direction which is normal to the curve in the sense of an anti-clockwise rotation through 90 degrees from the tangent vector **T** at the given point. (This is in the direction of **orthogonal_complement**(**T**).)

**self_intersect:** An indication of whether the offset curve self intersects; this is for information only.

Formal propositions:

**WR1:** The underlying curve shall be defined in two-dimensional space.

## 4.4.46 offset_curve_3d

An **offset_curve_3d** is a curve at a constant distance from a basis curve in three-dimensional space.

The underlying curve shall have a well-defined tangent direction at every point. In the case of a composite curve the transition code between each segment shall be **cont_same_gradient** or **cont_same_gradient_same_curvature**.

The offset curve at any point (parameter) on the basis curve is in the direction $\boldsymbol{V} \times \boldsymbol{T}$ where **V** is the fixed reference direction and **T** is the unit tangent to the **basis_curve**. For the offset direction to be well defined, **T** shall not at any point of the curve be in the same, or opposite, direction as **V**.

NOTE – The **offset_curve_3d** may differ in nature from the **basis_curve**; the offset of a non-self-intersecting curve can be self intersecting. Care should be taken to ensure that the offset to a continuous curve does not become discontinuous.

The **offset_curve_3d** takes its parametrisation from the **basis_curve**. The **offset_curve_3d** is parametrised as

$$\boldsymbol{\lambda}(u) = \mathbf{C}(u) + d\boldsymbol{V} \times \boldsymbol{T}.$$

Where **T** is the unit tangent vector to the basis curve $\mathbf{C}(u)$ at parameter value $u$, and $d$ is **distance**.

*EXPRESS* specification:

```
*)
ENTITY offset_curve_3d
  SUBTYPE OF (curve);
  basis_curve    : curve;
  distance       : length_measure;
  self_intersect : LOGICAL;
  ref_direction  : direction;
WHERE
  WR1 : (basis_curve.dim = 3) AND (ref_direction.dim = 3);
END_ENTITY;
(*
```

Attribute definitions:

**basis_curve:** The **curve** that is being offset.

**distance:** The distance of the offset curve from the basis curve. The distance may be positive, negative or zero.

**self_intersect:** An indication of whether the offset curve self intersects, this is for information only.

**ref_direction:** The **direction** used to define the direction of the **offset_curve** from the **basis_curve**.

Formal propositions:

**WR1:** Both the underlying curve and the reference direction shall be in three-dimensional space.

Informal propositions:

**IP1:** At no point on the curve shall **ref_direction** be parallel, or opposite to, the direction of the tangent vector.

### 4.4.47 curve_replica

A **curve_replica** is a replica of a curve in a different location. It is defined by referencing the parent curve and a transformation. The geometric form of the curve produced will be the same as the parent curve, but, where the transformation includes scaling, the dimensions will differ. The curve replica takes its parametric range and parametrisation directly from the parent curve. Where the parent curve is a curve on surface, the replica will not in general share the property of lying on the surface.

*EXPRESS* specification:

```
*)
ENTITY curve_replica
  SUBTYPE OF (curve);
  parent_curve   : curve;
  transformation : cartesian_transformation_operator;
WHERE
  WR1: transformation.dim = parent_curve.dim;
  WR2: acyclic_curve_replica (SELF, parent_curve);
END_ENTITY;
(*
```

Attribute definitions:

**parent_curve:** The curve that is being copied.

**transformation:** The cartesian transformation operator which defines the location of the curve replica. This transformation may include scaling.

Formal propositions:

**WR1:** The coordinate space dimensionality of the transformation attribute shall be the same as that of the **parent_curve**.

**WR2:** A **curve_replica** shall not participate in its own definition.

## 4.4.48 surface

See 3.1 for definition. A **surface** can be envisioned as a set of connected points in 3-dimensional space which is always locally 2-dimensional, but need not be a manifold. A surface shall not be a single point or in part, or entirely, a curve.

Each surface has a parametric representation of the form

$$\boldsymbol{\sigma}(u, v),$$

where $u$ and $v$ are independent dimensionless parameters. The unit normal **N**, at any point on the surface, is given by the equation

$$\mathbf{N}(u, v) = \langle \frac{\partial \boldsymbol{\sigma}}{\partial u} \times \frac{\partial \boldsymbol{\sigma}}{\partial v} \rangle$$

*EXPRESS* specification:

```
*)
ENTITY surface
  SUPERTYPE OF (ONEOF(elementary_surface, swept_surface, bounded_surface,
                      offset_surface, surface_replica))
  SUBTYPE OF (geometric_representation_item);
END_ENTITY;
(*
```

Informal propositions:

**IP1:** A **surface** has non-zero area.

**IP2:** A **surface** is arcwise connected.

### 4.4.49 elementary_surface

An elementary surface is a simple analytic surface with defined parametric representation.

*EXPRESS* specification:

```
*)
ENTITY elementary_surface
  SUPERTYPE OF (ONEOF(plane, cylindrical_surface, conical_surface,
                    spherical_surface, toroidal_surface))
  SUBTYPE OF (surface);
  position : axis2_placement_3d;
END_ENTITY;
(*
```

Attribute definitions:

**position:** The position and orientation of the surface. This attribute is used in the definition of the parametrisation of the surface.

### 4.4.50 plane

A **plane** is an unbounded surface with a constant normal. A **plane** is defined by a point on the plane and the normal direction to the plane. The data is to be interpreted as follows:

$$\mathbf{C} = \text{position.location}$$
$$\mathbf{x} = \text{position.p[1]}$$
$$\mathbf{y} = \text{position.p[2]}$$
$$\mathbf{z} = \text{position.p[3]} = \text{normal to plane}$$

and the surface is parametrised as

$$\boldsymbol{\sigma}(u,v) = \mathbf{C} + \mathbf{x}u + \mathbf{y}v$$

where the parametrisation range is $-\infty < u, v < \infty$. In the above parametrisation, the length unit for the unit vectors **x** and **y** is derived from the context of the plane.

*EXPRESS* specification:

```
*)
ENTITY plane
SUBTYPE OF (elementary_surface);
END_ENTITY;
(*
```

Attribute definitions:

**SELF\elementary_surface.position:** The location and orientation of the surface. This attribute is inherited from the **elementary_surface** supertype.

**position.location:** A point in the plane.

**position.p[3]:** This direction, which is equal to **position.axis**, defines the normal to the plane.

### 4.4.51 cylindrical_surface

A **cylindrical_surface** is a surface at a constant distance (the **radius**)from a straight line. A **cylindrical_surface** is defined by its radius and its orientation and location. The data is to be interpreted as follows:

$$\begin{aligned} \mathbf{C} &= \text{position.location} \\ \mathbf{x} &= \text{position.p[1]} \\ \mathbf{y} &= \text{position.p[2]} \\ \mathbf{z} &= \text{position.p[3]} \\ R &= \text{radius} \end{aligned}$$

and the surface is parametrised as

$$\boldsymbol{\sigma}(u,v) = \mathbf{C} + R((\cos u)\mathbf{x} + (\sin u)\mathbf{y}) + v\mathbf{z}$$

where the parametrisation range is $0 \leq u \leq 360$ degrees and $-\infty < v < \infty$. In the above parametrisation, the length unit for the unit vector **z** is equal to that of the **radius**.

In the placement coordinate system defined above, the surface is represented by the equation $\mathcal{S} = 0$, where

$$\mathcal{S}(x,y,z) = x^2 + y^2 - R^2$$

The positive direction of the normal to the surface at any point on the surface is given by

$$(\mathcal{S}_x, \mathcal{S}_y, \mathcal{S}_z).$$

The unit normal is given by

$$\mathbf{N}(u,v) = (\cos u)\mathbf{x} + (\sin u)\mathbf{y}.$$

The sense of this normal is away from the axis of the cylinder.

*EXPRESS* specification:

```
*)
ENTITY
cylindrical_surface
  SUBTYPE OF (elementary_surface);
  radius : positive_length_measure;
END_ENTITY;
(*
```

Attribute definitions:

**SELF\elementary_surface.position:** The location and orientation of the cylinder.

**position.location:** A point on the axis of the cylinder.

**position.p[3]:** The direction of the axis of the cylinder.

**radius:** The radius of the cylinder.

## 4.4.52 conical_surface

A **conical_surface** is a surface which could be produced by revolving a line in 3-dimensional space about any intersecting line. A **conical_surface** is defined by the semi-angle, the location and orientation and by the radius of the cone in the plane passing through the location point **C** normal to the cone axis.

> NOTE – This form of representation is designed to provide the greatest geometric precision for those parts of the surface which are close to the location point C. For this reason the apex should only be selected as location point if the region of the surface close to the apex is of interest.



The data is to be interpreted as follows:

$$\begin{aligned}
\mathbf{C} &= \text{position.location} \\
\mathbf{x} &= \text{position.p[1]} \\
\mathbf{y} &= \text{position.p[2]} \\
\mathbf{z} &= \text{position.p[3]} \\
R &= \text{radius} \\
\alpha &= \text{semi_angle}
\end{aligned}$$

and the surface is parametrised as

$$\boldsymbol{\sigma}(u,v) = \mathbf{C} + (R + v\tan\alpha)((\cos u)\mathbf{x} + (\sin u)\mathbf{y}) + v\mathbf{z}$$

where the parametrisation range is $0 \leq u \leq 360$ degrees and $-\infty < v < \infty$. In the above parametrisation the length unit for the unit vector **z** is equal to that of the **radius**.

In the placement coordinate system defined above, the surface is represented by the equation $\mathcal{S} = 0$, where

$$\mathcal{S}(x,y,z) = x^2 + y^2 - (R + z\tan\alpha)^2$$

The positive direction of the normal to the surface at any point on the surface is given by

$$(\mathcal{S}_x, \mathcal{S}_y, \mathcal{S}_z).$$

The unit normal is given by

$$\mathbf{N}(u,v) = \frac{(\cos u)\mathbf{x} + (\sin u)\mathbf{y} - (\tan\alpha)\mathbf{z}}{\sqrt{1 + (\tan\alpha)^2}}.$$

The sense of the normal is away from the axis of the cone. If the radius is zero, the cone apex is at the point $(0,0,0)$ in the placement coordinate system (i.e., at position.location).

*EXPRESS* specification:

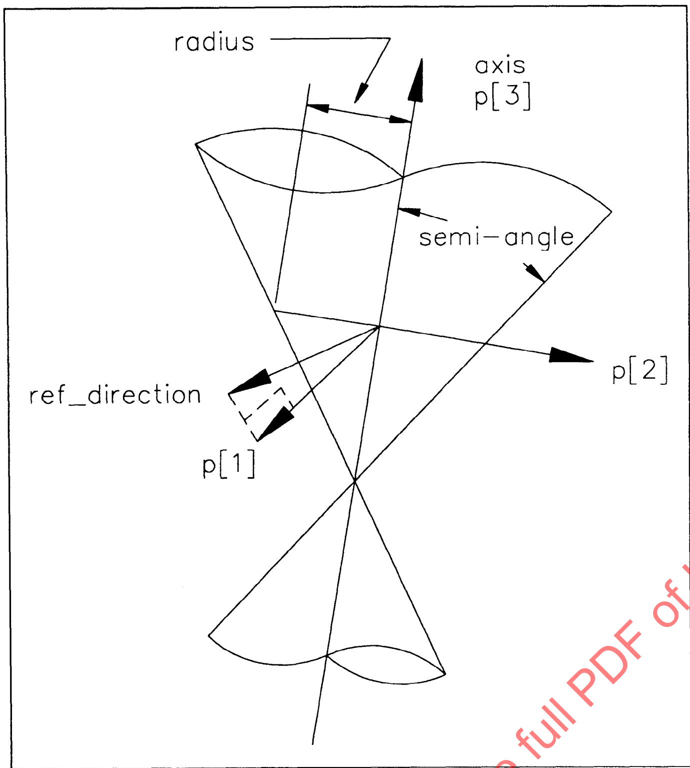


**Figure 9 – Conical surface**

```
*)
ENTITY
conical_surface
  SUBTYPE OF (elementary_surface);
  radius     : length_measure;
  semi_angle : plane_angle_measure;
WHERE
  WR1: radius >= 0.0;
END_ENTITY;
(*
```

Attribute definitions:

**SELF\elementary_surface.position:** The location and orientation of the surface.

**position.location:** The location point on the axis of the cone.

**position.p[3]:** The direction of the axis of the cone.

**radius:** The radius of the circular curve of intersection between the cone and a plane perpendicular to the axis of the cone passing through the location point (i.e., position.location).

**semi_angle:** The cone semi-angle.

NOTE – See figure 9 for illustration of the attributes.

Formal propositions:

**WR1:** The radius shall not be negative.

Informal propositions:

**IP1:** The semi-angle shall be between 0 and 90 degrees.

### 4.4.53 spherical_surface

A spherical surface is a surface which is at a constant distance (the **radius**) from a central point. A **spherical_surface** is defined by the radius and the location and orientation of the surface.

The data is to be interpreted as follows:

$$\begin{aligned} \mathbf{C} &= \text{position.location (centre)} \\ \mathbf{x} &= \text{position.p[1]} \\ \mathbf{y} &= \text{position.p[2]} \\ \mathbf{z} &= \text{position.p[3]} \\ R &= \text{radius} \end{aligned}$$

and the surface is parametrised as

$$\boldsymbol{\sigma}(u,v) = \mathbf{C} + R\cos v((\cos u)\mathbf{x} + (\sin u)\mathbf{y}) + R(\sin v)\mathbf{z}$$

where the parametrisation range is $0 \leq u \leq 360$ degrees and $-90 \leq v \leq 90$ degrees.

In the placement coordinate system defined above, the surface is represented by the equation $\mathcal{S} = 0$, where

$$\mathcal{S}(x,y,z) = x^2 + y^2 + z^2 - R^2.$$

The positive direction of the normal to the surface at any point on the surface is given by

$$(\mathcal{S}_x, \mathcal{S}_y, \mathcal{S}_z).$$

The unit normal is given by

$$\mathrm{N}(u,v) = \cos v((\cos u)\mathbf{x} + (\sin u)\mathbf{y}) + (\sin v)\mathbf{z},$$

that is, it is directed away from the centre of the sphere.

*EXPRESS* specification:

```
*)
ENTITY spherical_surface
  SUBTYPE OF (elementary_surface);
  radius    : positive_length_measure;
END_ENTITY;
(*
```

Attribute definitions:

**SELF\elementary_surface.position:** The location and orientation of the surface. **position.location** is the centre of the sphere.

**radius:** The radius of the sphere.

## 4.4.54 toroidal_surface

A **toroidal_surface** is a surface which could be produced by revolving a circle about a line in its plane. The radius of the circle being revolved is referred to here as the **minor_radius** and the **major_radius** is the distance from the centre of this circle to the axis of revolution. A **toroidal_surface** is defined by the major and minor radii and the position and orientation of the surface.

The data is to be interpreted as follows:

$$\begin{aligned} \mathbf{C} &= \text{position.location} \\ \mathbf{x} &= \text{position.p[1]} \\ \mathbf{y} &= \text{position.p[2]} \\ \mathbf{z} &= \text{position.p[3]} \\ R &= \text{major\_radius} \\ r &= \text{minor\_radius} \end{aligned}$$

and the surface is parametrised as

$$\boldsymbol{\sigma}(u, v) = \mathbf{C} + (R + r\cos v)((\cos u)\mathbf{x} + (\sin u)\mathbf{y}) + r(\sin v)\mathbf{z}$$

where the parametrisation range is $0 \leq u, v \leq 360$ degrees.

In the placement coordinate system defined above, the surface is represented by the equation $\mathcal{S} = 0$, where

$$\mathcal{S}(x, y, z) = x^2 + y^2 + z^2 - 2R\sqrt{x^2 + y^2} - r^2 + R^2.$$

The positive direction of the normal to the surface at any point on the surface is given by

$$(\mathcal{S}_x, \mathcal{S}_y, \mathcal{S}_z).$$

The unit normal is given by

$$\mathbf{N}(u, v) = \cos v((\cos u)\mathbf{x} + (\sin u)\mathbf{y}) + (\sin v)\mathbf{z}.$$

The sense of this normal is away from the nearest point on the circle of radius R with centre **C**. A manifold surface will be produced if the major radius is greater than the minor radius. If this condition is not fulfilled, the resulting surface will be self-intersecting.

*EXPRESS* specification:

```
*)
ENTITY toroidal_surface
  SUBTYPE OF (elementary_surface);
  major_radius : positive_length_measure;
  minor_radius : positive_length_measure;
END_ENTITY;
(*
```

Attribute definitions:

**SELF\elementary_surface.position:** The location and orientation of the surface. **position.location** is the central point of the torus.

**major_radius:** The major radius of the torus.

**minor_radius:** The minor radius of the torus.

### 4.4.55 degenerate_toroidal_surface

A **degenerate_toroidal_surface** is a special subtype of a **toroidal_surface** in which the **minor_radius** is greater than the **major_radius**. In this subtype the parametric range is restricted in order to define a manifold surface which is either the inner 'lemon-shaped' surface, or the outer 'apple-shaped' portion of the self-intersecting surface defined by the supertype.

The data is to be interpreted as follows:

$$\begin{aligned}\mathbf{C} &= \text{position.location} \\ \mathbf{x} &= \text{position.p[1]} \\ \mathbf{y} &= \text{position.p[2]} \\ \mathbf{z} &= \text{position.p[3]} \\ R &= \text{major\_radius} \\ r &= \text{minor\_radius}\end{aligned}$$

and the surface is parametrised as

$$\boldsymbol{\sigma}(u,v) = \mathbf{C} + (R + r\cos v)((\cos u)\mathbf{x} + (\sin u)\mathbf{y}) + r(\sin v)\mathbf{z}$$

where the parametrisation range is :

If **select_outer** = .TRUE. :
$0 \le u \le 360$ degrees.
$-\phi \le v \le \phi$ degrees.
If **select_outer** = .FALSE. :
$0 \le u \le 360$ degrees.
$\phi \le v \le 360 - \phi$ degrees.
Where $\phi$ degrees is the angle given by $r\cos\phi = -R$.

*EXPRESS* specification:

```
*)
ENTITY degenerate_toroidal_surface
  SUBTYPE OF (toroidal_surface);
  select_outer : BOOLEAN;
WHERE
  WR1: major_radius <   minor_radius;
END_ENTITY;
(*
```

Attribute definitions:

**select_outer:** A BOOLEAN flag used to distinguish between the two portions of the **degenerate_toroidal_surface**. If **select_outer** is true, the outer portion of the surface is selected and a closed 'apple-shaped' surface is defined. If **select_outer** is false, the inner portion is selected to define a closed 'lemon-shaped' axi-symmetric surface.

Formal propositions:

**WR1:** The major radius shall be less than the minor radius.

### 4.4.56 swept_surface

A **swept_surface** is one that is constructed by sweeping a curve along another curve.

*EXPRESS* specification:

```
*)
ENTITY swept_surface
  SUPERTYPE OF (ONEOF(surface_of_linear_extrusion, surface_of_revolution))
  SUBTYPE OF (surface);
  swept_curve : curve;
END_ENTITY;
(*
```

Attribute definitions:

**swept_curve:** The curve to be swept in defining the surface. If the swept curve is a pcurve, it is the image of this curve in 3D geometric space which is swept, not the parameter space curve.

### 4.4.57 surface_of_linear_extrusion

This surface is a simple swept surface or a generalised cylinder obtained by sweeping a curve in a given direction. The parametrisation is as follows, where the curve has a parametrisation $\boldsymbol{\lambda}(u)$:

$$\mathbf{V} = \text{extrusion\_axis}$$
$$\boldsymbol{\sigma}(u,v) = \boldsymbol{\lambda}(u) + v\boldsymbol{V}$$

The parametrisation range for $v$ is $-\infty < v < \infty$ and for $u$ is defined by the curve parametrisation.

*EXPRESS* specification:

```
*)
ENTITY surface_of_linear_extrusion
  SUBTYPE OF (swept_surface);
  extrusion_axis      : vector;
END_ENTITY;
(*
```

Attribute definitions:

**extrusion_axis:** The direction of extrusion, the magnitude of this vector determines the parametrisation.

**SELF\swept_surface.swept_curve:** The curve to be swept.

Informal propositions:

**IP1:** The surface shall not self-intersect.

### 4.4.58 surface_of_revolution

A **surface_of_revolution** is the surface obtained by rotating a curve one complete revolution about an axis.

The data shall be interpreted as below.

The parametrisation is as follows, where the curve has a parametrisation $\boldsymbol{\lambda}(v)$:

$$\begin{aligned} \mathbf{C} &= \text{position.location} \\ \mathbf{V} &= \text{position.z} \\ \boldsymbol{\sigma}(u,v) &= \mathbf{C} + (\boldsymbol{\lambda}(v) - \mathbf{C})\cos u + ((\boldsymbol{\lambda}(v) - \mathbf{C}) \cdot \mathbf{V})\mathbf{V}(1 - \cos u) + \mathbf{V} \times (\boldsymbol{\lambda}(v) - \mathbf{C})\sin u \end{aligned}$$

In order to produce a single-valued surface with a complete revolution, the curve shall be such that when expressed in a cylindrical coordinate system $(r, \phi, z)$ centred at $\mathbf{C}$ with axis $\mathbf{V}$, no two distinct parametric points on the curve shall have the same values for $(r, z)$.

NOTE – In this context a single valued surface is interpreted as one for which the mapping, from the interior of the rectangle in parameter space corresponding to its parametric range, to geometric space, defined by the surface equation, is one-to-one.

For a surface of revolution the parametric range is $0 \leq u \leq 360$ degrees.

The parameter range for $v$ is defined by the referenced curve.

NOTE – The geometric shape of the surface is not dependent upon the curve parametrisation.

*EXPRESS* specification:

```
*)
ENTITY surface_of_revolution
  SUBTYPE OF (swept_surface);
  axis_position        : axis1_placement;
DERIVE
  axis_line : line := line(axis_position.location,
                           vector(axis_position.z, 1.0));
END_ENTITY;
(*
```



Attribute definitions:

**axis_position:** A point on the axis of revolution and the direction of the axis of revolution.

**SELF\swept_surface.swept_curve:** The curve that is revolved about the axis line.

**axis_line:** The line coinciding with the axis of revolution.

Informal propositions:

**IP1:** The surface shall not self-intersect.

**IP2:** The **swept_curve** shall not be coincident with the **axis_line** for any finite part of its length.

### 4.4.59 bounded_surface

A **bounded_surface** is a surface of finite area with identifiable boundaries.

*EXPRESS* specification:

```
*)
ENTITY bounded_surface
  SUPERTYPE OF (ONEOF(b_spline_surface, rectangular_trimmed_surface,
                     curve_bounded_surface, rectangular_composite_surface))
  SUBTYPE OF (surface);
END_ENTITY;
(*
```

Informal propositions:

**IP1:** A **bounded_surface** has a finite non-zero surface area,

**IP2:** A **bounded_surface** has boundary curves.



### 4.4.60 b_spline_surface

A **b_spline_surface** is a general form of rational or polynomial parametric surface which is represented by control points, basis functions, and possibly, weights. As with the corresponding curve entity it has some special subtypes where some of the data can be derived.

NOTES

1 – Identification of B-spline surface default values and subtypes is important for performance considerations and for efficiency issues in performing computations.

2 – A B-spline is *rational* if and only if the weights are not all identical. If it is polynomial, the weights may be defaulted to all being 1.

3 – In the case where the B-spline surface is uniform, quasi-uniform or piecewise Bézier, the knots and knot multiplicities may be defaulted (i.e., non-existent in the data as specified by the attribute definitions). When the knots are defaulted, a difference of 1.0 between separate knots is assumed, and the effective parameter range for the resulting surface starts from 0.0. These defaults are provided by the subtypes.

4 – The knots and knot multiplicities shall not be defaulted in the non-uniform case.

5 – The defaulting of weights and knots are done independently of one another.

The data is to be interpreted as follows:

a) The symbology used here is:

$$\begin{aligned} K1 &= \text{upper\_index\_on\_u\_control\_points} \\ K2 &= \text{upper\_index\_on\_v\_control\_points} \\ \mathbf{P}_{ij} &= \text{control\_points} \\ w_{ij} &= \text{weights} \\ d1 &= \text{u\_degree} \\ d2 &= \text{v\_degree} \end{aligned}$$

b) The control points are ordered as

$$\mathbf{P}_{00}, \mathbf{P}_{01}, \mathbf{P}_{02}, \cdots\cdots, \mathbf{P}_{K1(K2-1)}, \mathbf{P}_{K1K2}$$

The weights, in the case of the rational subtype, are ordered similarly.

c) For each parameter, $s = u$ or $v$, if $k$ is the upper index on the control points and $d$ is the degree for $s$, the knot array is an array of $(k + d + 2)$ real numbers $[s_{-d}, ..., s_{k+1}]$, such that for all indices j in $[-d, k]$, $s_j <= s_{j+1}$. This array is obtained from the appropriate **knots_data** list by repeating each multiple knot according to the multiplicity.
$N_i^d$, the $i$th normalised B-spline basis function of degree $d$, is defined on the subset $[s_{i-d}, ..., s_{i+1}]$ of this array.

d) Let $L$ denote the number of distinct values amongst the knots in the knot list; $L$ will be referred to as the 'upper index on knots'. Let $m_j$ denote the multiplicity (i.e., number of repetitions) of the $j$th distinct knot value. Then:

$$\sum_{i=1}^{L} m_i = d + k + 2$$

All knot multiplicities except the first and the last shall be in the range $1 \ldots d$; the first and last may have a maximum value of $d + 1$. In evaluating the basis functions, a knot $u$ of, e.g., multiplicity 3 is interpreted as a sequence $u, u, u,$ in the knot array.

e) The surface form is used to identify specific quadric surface types (which shall have degree two), ruled surfaces and surfaces of revolution. As with the **b_spline_curve**, the form number is informational only and the spline data takes precedence.

f) The surface is to be interpreted as follows: In the polynomial case the surface is given by the equation:

$$\boldsymbol{\sigma}(u, v) = \sum_{i=0}^{K1} \sum_{j=0}^{K2} \mathbf{P}_{ij} N_i^{d1}(u) N_j^{d2}(v)$$

In the rational case the surface equation is:

$$\boldsymbol{\sigma}(u, v) = \frac{\sum_{i=0}^{K1} \sum_{j=0}^{K2} w_{ij} \mathbf{P}_{ij} N_i^{d1}(u) N_j^{d2}(v)}{\sum_{i=0}^{K1} \sum_{j=0}^{K2} w_{ij} N_i^{d1}(u) N_j^{d2}(v)}$$

NOTE – Definitions of the B-spline basis functions, $\mathbf{P}_{ij} N_i^{d1}(u), N_j^{d2}(v)$, can be found in [E-1, E-2, E-3]. It should be noted that there is a difference in terminology between these references.

*EXPRESS* specification:

```
*)
ENTITY b_spline_surface
  SUPERTYPE OF (ONEOF(b_spline_surface_with_knots, uniform_surface,
                     quasi_uniform_surface, bezier_surface)
                     ANDOR rational_b_spline_surface)
  SUBTYPE OF (bounded_surface);
  u_degree              : INTEGER;
  v_degree              : INTEGER;
  control_points_list   : LIST [2:?] OF
                             LIST [2:?] OF cartesian_point;
  surface_form          : b_spline_surface_form;
  u_closed              : LOGICAL;
  v_closed              : LOGICAL;
  self_intersect        : LOGICAL;
DERIVE
  u_upper              : INTEGER := SIZEOF(control_points_list) - 1;
  v_upper              : INTEGER := SIZEOF(control_points_list[1]) - 1;
  control_points       : ARRAY [0:u_upper] OF ARRAY [0:v_upper] OF
                         cartesian_point
                      := make_array_of_array(control_points_list,
                                             0,u_upper,0,v_upper);
WHERE
  WR1: ('GEOMETRY_SCHEMA.UNIFORM_SURFACE' IN TYPEOF(SELF)) OR
       ('GEOMETRY_SCHEMA.QUASI_UNIFORM_SURFACE' IN TYPEOF(SELF)) OR
       ('GEOMETRY_SCHEMA.BEZIER_SURFACE' IN TYPEOF(SELF)) OR
       ('GEOMETRY_SCHEMA.B_SPLINE_SURFACE_WITH_KNOTS' IN TYPEOF(SELF));
END_ENTITY;
(*
```



Attribute definitions:

**u_degree:** Algebraic degree of basis functions in u.

**v_degree:** Algebraic degree of basis functions in v.

**control_points_list:** This is a list of lists of control points.

**surface_form:** Indicator of special surface types. (See 4.3.5.)

**u_closed:** Indication of whether the surface is closed in the **u** direction; this is for information only.

**v_closed:** Indication of whether the surface is closed in the **v** direction; this is for information only.

**self_intersect:** Flag to indicate whether, or not, surface is self-intersecting; this is for information only.

**u_upper:** Upper index on control points in u direction.

**v_upper:** Upper index on control points in v direction.

**control_points:** Array (two-dimensional) of control points defining surface geometry. This array is constructed from the control points list.

Formal propositions:

**WR1:** Any instantiation of this entity shall include one of the subtypes **b_spline_surface_with_knots**, **uniform_surface**, **quasi_uniform_surface**, or **bezier_surface**.

## 4.4.61 b_spline_surface_with_knots

This is a B-spline surface in which the knot values are explicitly given. This subtype shall be used to represent non-uniform B-spline surfaces, and may also be used for other knot types.

All knot multiplicities except the first and the last shall be in the range $1 \dots d$; the first and last may have a maximum value of $d + 1$.

In evaluating the basis functions, a knot $u$ of, e.g., multiplicity 3 is interpreted as a sequence $u, u, u$, in the knot array.

*EXPRESS* specification:

```
*)
ENTITY b_spline_surface_with_knots
  SUBTYPE OF (b_spline_surface);
  u_multiplicities    : LIST [2:?] OF INTEGER;
  v_multiplicities    : LIST [2:?] OF INTEGER;
  u_knots             : LIST [2:?] OF parameter_value;
  v_knots             : LIST [2:?] OF parameter_value;
  knot_spec           : knot_type;
DERIVE
  knot_u_upper        : INTEGER := SIZEOF(u_knots);
  knot_v_upper        : INTEGER := SIZEOF(v_knots);
WHERE
   WR1: constraints_param_b_spline(SELF\b_spline_surface.u_degree,
                  knot_u_upper, SELF\b_spline_surface.u_upper,
                              u_multiplicities, u_knots);
   WR2: constraints_param_b_spline(SELF\b_spline_surface.v_degree,
                  knot_v_upper, SELF\b_spline_surface.v_upper,
                              v_multiplicities, v_knots);
   WR3: SIZEOF(u_multiplicities) = knot_u_upper;
   WR4: SIZEOF(v_multiplicities) = knot_v_upper;
END_ENTITY;
(*
```

Attribute definitions:

**u_multiplicities:** The multiplicities of the knots in the u parameter direction.

**v_multiplicities:** The multiplicities of the knots in the v parameter direction.

**u_knots:** The list of the distinct knots in the u parameter direction.

**v_knots:** The list of the distinct knots in the v parameter direction.

**knot_spec:** The description of the knot type.

**knot_u_upper:** The number of distinct knots in the u parameter direction.

**knot_v_upper:** The number of distinct knots in the v parameter direction.

**SELF\b_spline_surface.u_degree:** Algebraic degree of basis functions in u.

**SELF\b_spline_surface.v_degree:** Algebraic degree of basis functions in v.

**SELF\b_spline_surface.control_points_list:** This is a list of lists of control points.

**SELF\b_spline_surface.surface_form:** Indicator of special surface types. (See 4.3.5.)

**SELF\b_spline_surface.u_closed:** Indication of whether the surface is closed in the **u** direction; this is for information only.

**SELF\b_spline_surface.v_closed:** Indication of whether the surface is closed in the **v** direction; this is for information only.

**SELF\b_spline_surface.self_intersect:** Flag to indicate whether, or not, surface is self-intersecting; this is for information only.

**SELF\b_spline_surface.u_upper:** Upper index on control points in u direction.

**SELF\b_spline_surface.v_upper:** Upper index on control points in v direction.

**SELF\b_spline_surface.control_points:** Array (two-dimensional) of control points defining surface geometry. This array is constructed from the control points list.

Formal propositions:

**WR1: constraints_param_b_spline** returns TRUE when the parameter constraints are verified for the **u**-direction.

**WR2: constraints_param_b_spline** returns TRUE when the parameter constraints are verified for the **v**-direction.

**WR3:** The number of **u_multiplicities** shall be the same as the number of **u_knots**.

**WR4:** The number of **v_multiplicities** shall be the same as the number of **v_knots**.

## 4.4.62 uniform_surface

This is a special subtype of **b_spline_surface** in which the knots are evenly spaced. Suitable default values for the knots and knot multiplicities can be derived in this case.

A B-spline is *uniform* if and only if all knots are of multiplicity 1 and they differ by a positive constant from the preceding knot. In this subtype the knot spacing is 1.0, starting from $-degree$.

EXPRESS specification:

```
*)
ENTITY uniform_surface
  SUBTYPE OF (b_spline_surface);
END_ENTITY;
(*
```

NOTE – If explicit knot values for the surface are required, they can be derived as follows:

ku_up := SELF\b_spline_surface.u_upper +SELF\b_spline_surface.u_degree + 2;
kv_up := SELF\b_spline_surface.v_upper + SELF\b_spline_surface.v_degree + 2;

**ku_up** is the value required for the upper index on the knot and knot multiplicity lists in the u direction. This is computed from the degree and the number of control points in this direction.
**kv_up** is the value required for the upper index on the knot and knot multiplicity lists in the v direction. This is computed from the degree and the number of control points in this direction. The knot multiplicities and knots in the $u$ and $v$ parameter directions are then given by the function calls:
default_b_spline_knot_mult(SELF\b_spline_surface.u_degree, ku_up, uniform_knots)
default_b_spline_knots(SELF\b_spline_surface.u_degree,ku_up, uniform_knots)
default_b_spline_knot_mult(SELF\b_spline_surface.v_degree, kv_up, uniform_knots)
default_b_spline_knots(SELF\b_spline_surface.v_degree,kv_up, uniform_knots)

## 4.4.63 quasi_uniform_surface

This is a special subtype of **b_spline_surface** in which the knots are evenly spaced, and except for the first and last, have multiplicity 1. Suitable default values for the knots and knot multiplicities are derived in this case.

A B-spline is *quasi-uniform* if and only if the knots are of multiplicity (degree+1) at the ends, of multiplicity 1 elsewhere, and they differ by a positive constant from the preceding knot. In this subtype the knot spacing is 1.0, starting from 0.0.

EXPRESS specification:

```
*)
ENTITY quasi_uniform_surface
  SUBTYPE OF (b_spline_surface);
END_ENTITY;
(*
```

NOTE – If explicit knot values for the surface are required, they can be derived as follows:

ku_up := SELF\b_spline_surface.u_upper - SELF\b_spline_surface.u_degree + 2;
kv_up := SELF\b_spline_surface.v_upper - SELF\b_spline_surface.v_degree + 2;

**ku_up** is the value required for the upper index on the knot and knot multiplicity lists in the u direction. This is computed from the degree and the number of control points in this direction.
**kv_up** is the value required for the upper index on the knot and knot multiplicity lists in the v direction. This is computed from the degree and the number of control points in this direction. The knot multiplicities and knots in the $u$ and $v$ parameter directions are then given by the function calls:

default_b_spline_knot_mult(SELF\b_spline_surface.u_degree, ku_up, quasi_uniform_knots)
default_b_spline_knots(SELF\b_spline_surface.u_degree,ku_up, quasi_uniform_knots)
default_b_spline_knot_mult(SELF\b_spline_surface.v_degree, kv_up, quasi_uniform_knots)
default_b_spline_knots(SELF\b_spline_surface.v_degree,kv_up, quasi_uniform_knots)



## 4.4.64 bezier_surface

This is a special type of surface which can be represented as a subtype of **b_spline_surface** in which the knots are evenly spaced and have high multiplicities. Suitable default values for the knots and knot multiplicities are derived in this case. In this subtype the knot spacing is 1.0, starting from 0.0.

*EXPRESS* specification:

```
*)
ENTITY bezier_surface
  SUBTYPE OF (b_spline_surface);
END_ENTITY;
(*
```

NOTE – If explicit knot values for the surface are required, they can be derived as follows:

$$ku\_up := \frac{SELF\backslash b\_spline\_surface.u\_upper}{SELF\backslash b\_spline\_surface.u\_degree} + 1;$$

$$kv\_up := \frac{SELF\backslash b\_spline\_surface.v\_upper}{SELF\backslash b\_spline\_surface.v\_degree} + 1;$$

**ku_up** is the value required for the upper index on the knot and knot multiplicity lists in the u direction. This is computed from the degree and the number of control points in this direction.
**kv_up** is the value required for the upper index on the knot and knot multiplicity lists in the v direction. This is computed from the degree and the number of control points in this direction. The knot multiplicities and knots in the $u$ and $v$ parameter directions are then given by the function calls:

default_b_spline_knot_mult(SELF\b_spline_surface.u_degree, ku_up, bezier_knots)
default_b_spline_knots(SELF\b_spline_surface.u_degree,ku_up, bezier_knots)
default_b_spline_knot_mult(SELF\b_spline_surface.v_degree, kv_up, bezier_knots)
default_b_spline_knots(SELF\b_spline_surface.v_degree,kv_up, bezier_knots)

### 4.4.65 rational_b_spline_surface

A **rational_b_spline_surface** is a piecewise parametric rational surface described in terms of control points, associated weight values and basis functions. It is instantiated with any of the other subtypes of **b_spline_surface**, which provide explicit or implicit knot values from which the basis functions are defined.

The surface is to be interpreted as follows:

$$\boldsymbol{\sigma}(u,v) = \frac{\sum_{i=0}^{K1}\sum_{j=0}^{K2} w_{ij}\mathbf{P}_{ij}N_i^{d1}(u)N_j^{d2}(v)}{\sum_{i=0}^{K1}\sum_{j=0}^{K2} w_{ij}N_i^{d1}(u)N_j^{d2}(v)}$$

NOTE – See 4.4.60 for details of the symbology used in the above equation.

*EXPRESS* specification:

```
*)
ENTITY rational_b_spline_surface
  SUBTYPE OF (b_spline_surface);
  weights_data : LIST [2:?] OF
                   LIST [2:?] OF REAL;

DERIVE
  weights       : ARRAY [0:u_upper] OF
                    ARRAY [0:v_upper] OF REAL
                := make_array_of_array(weights_data,0,u_upper,0,v_upper);
WHERE
  WR1: (SIZEOF(weights_data) =
                        SIZEOF(SELF\b_spline_surface.control_points_list))
          AND (SIZEOF(weights_data[1]) =
                        SIZEOF(SELF\b_spline_surface.control_points_list[1]));
  WR2: surface_weights_positive(SELF);
END_ENTITY;
(*
```

Attribute definitions:

**weights_data:** The weights associated with the control points in the rational case.

**weights:** Array (two-dimensional) of weight values constructed from the **weights_data**.

Formal propositions:

**WR1:** The array dimensions for the weights shall be consistent with the control points data.

**WR2:** The weight value associated with each control point shall be greater than zero.

## 4.4.66 rectangular_trimmed_surface

The trimmed surface is a simple **bounded_surface** in which the boundaries are the constant parametric lines $u_1$ = u1, $u_2$ = u2, $v_1$ = v1 and $v_2$ = v2. All these values shall be within the parametric range of the referenced surface. Cyclic properties of the parameter range are assumed.

NOTES

1 – For example, 370 degrees is equivalent to 10 degrees, for those surfaces whose parametric form is defined using circular functions (sine and cosine). The rectangular trimmed surface inherits its parametrisation directly from the basis surface and has parameter ranges from 0 to $|u_2 - u_1|$ and 0 to $|v_2 - v_1|$. The derivation of the new parameters from the old uses the algorithm described in 4.4.34.

2 – If the surface is closed in a given parametric direction, the values of $u_2$ or $v_2$ may require to be increased by the cyclic range.

*EXPRESS* specification:

```
*)
ENTITY rectangular_trimmed_surface
  SUBTYPE OF (bounded_surface);
  basis_surface : surface;
  u1            : parameter_value;
  u2            : parameter_value;
  v1            : parameter_value;
  v2            : parameter_value;
  usense        : BOOLEAN;
  vsense        : BOOLEAN;
WHERE
  WR1: u1 <> u2;
  WR2: v1 <> v2;
  WR3: (('GEOMETRY_SCHEMA.ELEMENTARY_SURFACE' IN TYPEOF(basis_surface)) AND
        (NOT ('GEOMETRY_SCHEMA.PLANE' IN TYPEOF(basis_surface)))) OR
        ('GEOMETRY_SCHEMA.SURFACE_OF_REVOLUTION' IN TYPEOF(basis_surface)) OR
        (usense = (u2 > u1));
  WR4: (('GEOMETRY_SCHEMA.SPHERICAL_SURFACE' IN TYPEOF(basis_surface)) OR
        ('GEOMETRY_SCHEMA.TOROIDAL_SURFACE' IN TYPEOF(basis_surface))) OR
        (vsense = (v2 > v1));
END_ENTITY;
(*
```



Attribute definitions:

**basis_surface:** Surface being trimmed.

**u1:** First $u$ parametric value.

**u2:** Second $u$ parametric value.

**v1:** First $v$ parametric value.

**v2:** Second $v$ parametric value.

**usense:** Flag to indicate whether the direction of the first parameter of the trimmed surface agrees with or opposes the sense of $u$ in the basis surface.

**vsense:** Flag to indicate whether the direction of the second parameter of the trimmed surface agrees with or opposes the sense of $v$ in the basis surface.

Formal propositions:

**WR1:** **u1** and **u2** shall have different values.

**WR2:** **v1** and **v2** shall have different values.

**WR3:** With the exception of those surfaces closed in the $u$ parameter direction, **usense** shall be compatible with the ordered parameter values for $u$.

**WR4:** With the exception of those surfaces closed in the $v$ parameter direction, **vsense** shall be compatible with the ordered parameter values for $v$.

Informal propositions:

**IP1:** The domain of the trimmed surface shall be within the domain of the surface being trimmed.



### 4.4.67 curve_bounded_surface

The **curve_bounded_surface** is a parametric surface with curved boundaries defined by one or more boundary curves. One of these may be the outer boundary; any number of inner boundaries is permissible. The outer boundary may be defined implicitly as the natural boundary of the surface; this is indicated by the **implicit_outer** flag being true. In this case at least one inner boundary shall be defined. For certain types of closed surface (e.g. cylinder) it may not be possible to identify any given boundary as outer. The region of the **curve_bounded_surface** in the **basis_surface** is defined to be the portion of the basis surface in the direction of $\boldsymbol{N} \times \boldsymbol{T}$ from any point on the boundary, where **N** is the surface normal and **T** the boundary curve tangent vector at this point. The region so defined shall be arcwise connected.

*EXPRESS specification:*

```
*)
ENTITY curve_bounded_surface
  SUBTYPE OF (bounded_surface);
  basis_surface     : surface;
  boundaries        : SET [1:?] OF boundary_curve;
  implicit_outer    : BOOLEAN;
WHERE
  WR1: NOT(implicit_outer AND
           ('GEOMETRY_SCHEMA.OUTER_BOUNDARY_CURVE' IN TYPEOF(boundaries)));
  WR2: (NOT(implicit_outer)) OR
             ('GEOMETRY_SCHEMA.BOUNDED_SURFACE' IN TYPEOF(basis_surface));
  WR3: SIZEOF(QUERY(temp <* boundaries |
               'GEOMETRY_SCHEMA.OUTER_BOUNDARY_CURVE' IN
```

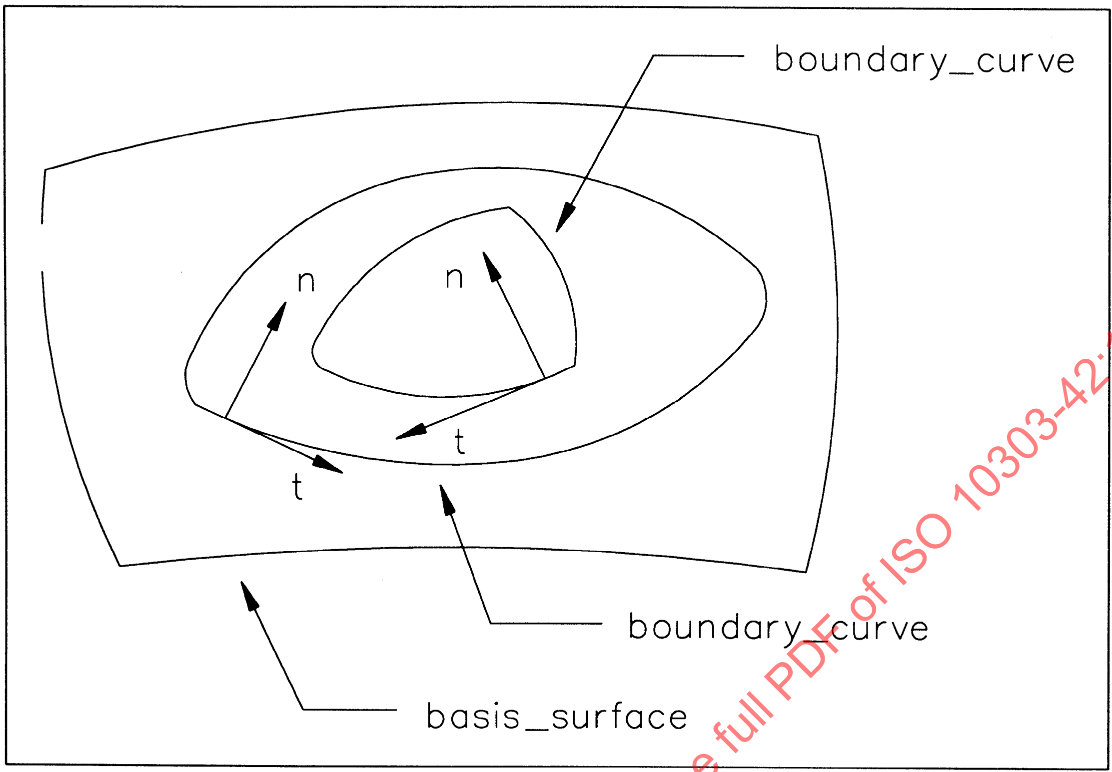


**Figure 10 – Curve bounded surface**

```
  WR4: SIZEOF(QUERY(temp <* boundaries |                TYPEOF(temp))) <= 1;
          (temp\composite_curve_on_surface.basis_surface [1] :<>:
                                             SELF.basis_surface))) = 0;
END_ENTITY;
(*
```

Attribute definitions:

**basis_surface:** The surface to be bounded.

**boundaries:** The bounding curves of the surface, other than the implicit outer boundary, if present. At most, one of these may be identified as an outer boundary by being of type **outer_boundary_curve.**

**implicit_outer:** A logical flag which, if true, indicates the natural boundary of the surface is used as an outer boundary.

NOTE – See figure 10 for interpretation of these attributes.

Formal propositions:

**WR1:** No explicit outer boundary shall be present when **implicit_outer** is TRUE.

**WR2:** The outer boundary shall only be implicitly defined if the **basis_surface** is bounded.

**WR3:** At most, one outer boundary curve shall be included in the list of boundaries

**WR4:** Each **boundary_curve** shall lie on the **basis_surface**. This is verified from the **basis_surface** attribute of the **composite_curve_on_surface** supertype for each element of the **boundaries** list.

Informal propositions:

**IP1:** Each curve in the set of **boundaries** shall be closed.

**IP2:** No two curves in the set of **boundaries** shall intersect.

**IP3:** At most one of the boundary curves may enclose any other boundary curve. If an **outer_boundary_curve** is designated, only that curve may enclose any other boundary curve.

### 4.4.68 boundary_curve

A **boundary_curve** is a type of bounded curve suitable for the definition of a surface boundary.

*EXPRESS* specification:

```
*)
ENTITY boundary_curve
  SUBTYPE OF (composite_curve_on_surface);
WHERE
  WR1: SELF\composite_curve.closed_curve;
END_ENTITY;
(*
```

Formal propositions:

**WR1:** The derived **closed_curve** attribute of the **composite_curve** supertype shall be TRUE.

### 4.4.69 outer_boundary_curve

This is a special sub-type of boundary curve which has the additional semantics of defining an outer boundary of a surface. No more than one such curve shall be included in the set of **boundaries** of a **curve_bounded_surface**.

*EXPRESS* specification:

```
*)
ENTITY outer_boundary_curve
  SUBTYPE OF (boundary_curve);
END_ENTITY;
(*
```

### 4.4.70 rectangular_composite_surface

This is a surface composed of a rectangular array of **n_u** by **n_v** segments or patches. Each segment shall be finite and topologically rectangular (i.e., it corresponds to a rectangle in parameter space). The segment shall be either a **b_spline_surface** or a **rectangular_trimmed_surface**. There shall be at least positional continuity between adjacent segments in both directions; the composite surface may be open or closed in the **u** direction and open or closed in the **v** direction.

For a particular segment $S_{ij}$ (= **segments[i][j]**):

– The preceding segment in the $u$ direction is $S_{(i-1)j}$ and the preceding segment in the $v$ direction is $S_{i(j-1)}$; similarly for following segments.

– If **segments[i][j].u_sense** is TRUE, the boundary of $S_{ij}$ where it adjoins $S_{(i+1)j}$ is that where the $u$ parameter (of the underlying bounded surface) is high.
If **segments[i][j].u_sense** is FALSE, it is at the low-$u$ boundary; similarly for the **v_sense** indicator.

– The $u$-parametrisation of $S_{ij}$ in the composite surface is from $i-1$ to $i$, mapped linearly from the parametrisation of the underlying bounded surface. If $U$ is the $u$-parameter for the **rectangular_composite_surface** and $u_{ij0} \leq u_{ij} \leq u_{ij1}$, is the $u$-parameter for **segments[i][j]**, these parameters are related by the equations:

$$U = (i-1) + \frac{u_{ij} - u_{ij0}}{u_{ij1} - u_{ij0}}, \quad u_{ij} = u_{ij0} + (U - (i-1))(u_{ij1} - u_{ij0}),$$

if **segments[i][j].u_sense** = TRUE;

$$U = i - \frac{u_{ij} - u_{ij0}}{u_{ij1} - u_{ij0}}, \quad u_{ij} = u_{ij0} - (U - i)(u_{ij1} - u_{ij0}),$$

if **segments[i][j].u_sense** = FALSE.
The $v$-parametrisation is obtained in a similar way.
Thus the composite surface has parametric range 0 to **n_u**, 0 to **n_v**.

– The degree of continuity of the joint between $S_{ij}$ and $S_{(i+1)j}$ is given by **segments[i][j].u_transition**.

For the last patch in a row $S_{(n_u)j}$ this may take the value **discontinuous**, if the composite surface is open in this direction; otherwise it is closed here, and the transition code relates to the continuity to $S_{1j}$; similarly for **v_transition**. **discontinuous** shall not occur elsewhere in the **segments surface_patch** transition codes.

<u>*EXPRESS* specification:</u>

```
*)
ENTITY rectangular_composite_surface
  SUBTYPE OF (bounded_surface);
  segments      : LIST [1:?] OF LIST [1:?] OF surface_patch;
```

```
DERIVE
  n_u : INTEGER := SIZEOF(segments);
  n_v : INTEGER := SIZEOF(segments[1]);
WHERE
  WR1: [] = QUERY (s <* segments | n_v <> SIZEOF (s));
  WR2: constraints_rectangular_composite_surface(SELF);
END_ENTITY;
(*
```

Attribute definitions:

**n_u:** The number of surface patches in the u parameter direction.

**n_v:** The number of surface patches in the v parameter direction.

**segments:** Rectangular array (represented by a list of list) of component surface patches. Each such patch contains information on the senses and transitions.

**segments[i][j].u_transition** refers to the state of continuity between **segments[i][j]** and **segments[i+1][j]**. The last column (**segments[n_u][j].u_transition**) may contain the value **discontinuous**, meaning that (that row of) the surface is not closed in the u direction; the rest of the list may not contain this value.

The last row (**segments[i][n_v].v_transition**) may contain the value **discontinuous**, meaning that (that column of) the surface is not closed in the v direction; the rest of the list may not contain this value.

Formal propositions:

**WR1:** Each sub-list in the **segments** list shall contain **n_v surface_patch**es.

**WR2:** Other constraints on the segments:

– that the component surfaces are all either rectangular trimmed surfaces or B-spline surfaces;

– that the **transition_code**s in the **segments** list do not contain the value **discontinuous** except for the last row or column; when this occurs, it indicates that the surface is not closed in the appropriate direction.

Informal propositions:

**IP1:** The senses of the component surfaces are as specified in the **u_sense** and **v_sense** attributes of each element of **segments**.

## 4.4.71 surface_patch

A surface patch is a bounded surface with additional transition and sense information which is used to define a **rectangular_composite_surface**.

*EXPRESS* specification:

```
*)
ENTITY surface_patch;
  parent_surface : bounded_surface;
  u_transition   : transition_code;
  v_transition   : transition_code;
  u_sense        : BOOLEAN;
  v_sense        : BOOLEAN;
INVERSE
  using_surfaces : BAG[1:?] OF rectangular_composite_surface FOR segments;
WHERE
  WR1: (NOT ('GEOMETRY_SCHEMA.CURVE_BOUNDED_SURFACE'
                IN TYPEOF(parent_surface)));
END_ENTITY;
(*
```

Attribute definitions:

**parent_surface:** The surface which defines the geometry and boundaries of the surface patch.

**u_transition:** The minimum state of geometric continuity along the second u boundary of the patch as it joins the first u boundary of its neighbour. In the case of the last patch, this defines the state of continuity between the first u boundary and last u boundary of the **composite_surface**.

**v_transition:** The minimum state of geometric continuity along the second v boundary of the patch as it joins the first v boundary of its neighbour. In the case of the last patch, this defines the state of continuity between the first v boundary and last v boundary of the **composite_surface**.

**u_sense:** This defines the relationship between the sense (increasing parameter value) of the patch and the sense of the **parent_surface**. If **u_sense** is TRUE, the first u boundary of the patch is the one where the parameter u takes its lowest value, it is the highest value boundary if sense is FALSE.

**v_sense:** This defines the relationship between the sense (increasing parameter value) of the patch and the sense of the **parent_surface**. If **v_sense** is TRUE, the first v boundary of the patch is the one where the parameter v takes its lowest value, it is the highest value boundary if sense is FALSE.

**using_surfaces:** The set of **rectangular_composite_surface**s which use this **surface_patch** in their definition. This set shall not be empty.

Formal propositions:

**WR1:** A curve bounded surface shall not be used to define a surface patch.

### 4.4.72 offset_surface

This is a procedural definition of a simple offset surface at a normal distance from the originating surface. **distance** may be positive, negative or zero to indicate the preferred side of the surface. The positive side and the resultant offset surface are defined as follows:

a) Define unit tangent vectors of the base surface in the $u$ and $v$ directions; denote these by $\boldsymbol{\sigma}_u$ and $\boldsymbol{\sigma}_v$.

b) Take the cross product, $\mathbf{N} = \boldsymbol{\sigma}_u \times \boldsymbol{\sigma}_v$, of these (which shall be linearly independent, or there is no offset surface). **N** shall be extended by continuity at singular points, if possible.

c) Normalise **N** to get a unit normal (to the surface) vector.

d) Move the offset distance (which may be zero) along that vector to find the point on the offset surface.

NOTE – The definition allows the **offset_surface** to be self-intersecting.

The offset surface takes its parametrisation directly from that of the basis surface, corresponding points having identical parameter values. The **offset_surface** is parametrised as

$$\boldsymbol{\sigma}(u, v) = \mathbf{S}(u, v) + d\mathbf{N}.$$

Where **N** is the unit normal vector to the basis surface $\mathbf{S}(u, v)$ at parameter values $(u, v)$, and $d$ is **distance**.

NOTE – Care should be taken when using this entity to ensure that the offset distance never exceeds the radius of curvature in any direction at any point of the basis surface. In particular, the surface should not contain any ridge or singular point.

*EXPRESS* specification:

```
*)
ENTITY offset_surface
  SUBTYPE OF (surface);
  basis_surface  : surface;
  distance       : length_measure;
  self_intersect : LOGICAL;
END_ENTITY;
(*
```

Attribute definitions:

**basis_surface:** The surface that is to be offset.

**distance:** The offset distance, which may be positive, negative or zero. A positive offset distance is measured in the direction of the surface normal.

**self_intersect:** Flag to indicate whether or not the surface is self-intersecting; this is for information only.

### 4.4.73 surface_replica

This defines a replica of an existing surface in a different location. It is defined by referencing the parent surface and a transformation which gives the new position and possible scaling. The original surface is not affected. The geometric characteristics of the surface produced will be identical to that of the parent surface, but, where the transformation includes scaling, the size may differ.

*EXPRESS* specification:

```
*)
ENTITY surface_replica
  SUBTYPE OF (surface);
  parent_surface : surface;
  transformation : cartesian_transformation_operator_3d;
WHERE
  WR1: acyclic_surface_replica(SELF, parent_surface);
END_ENTITY;
(*
```

Attribute definitions:

**parent_surface:** The surface that is being copied.

**transformation:** The **cartesian_transformation_operator_3d** which defines the location, orientation and scaling of the surface replica relative to that of the parent surface.

Formal propositions:

**WR1:** A **surface_replica** shall not participate in its own definition.

## 4.5 geometry_schema rule definitions

### 4.5.1 compatible_dimension

The rule **compatible_dimension** ensures that:

a) all **geometric_representation_items** are geometrically founded in one or more **geometric_representation_context** coordinate spaces;

b) when **geometric_representation_items** are geometrically founded together in a coordinate space, they have the same coordinate space **dimension_count** by ensuring that each matches the **dimension_count** of the coordinate space in which it is geometrically founded.

NOTE – Two-dimensional **geometric_representation_items** that are geometrically founded in a **geometric_representation_context** are only geometrically founded in **geometric_representation_contexts** with a **coordinate_space_dimension** of 2. All geometric_representation_items founded in such a context are two-dimensional. All other values of **dimension_count** behave similarly.

*EXPRESS* specification:

```
*)
RULE compatible_dimension FOR
  (cartesian_point,
  direction,
  representation_context,
  geometric_representation_context);
WHERE
```

```
  -- ensure that the count of coordinates of each cartesian_point
  -- matches the coordinate_space_dimension of each geometric_context in
  -- which it is geometrically_founded
  WR1: SIZEOF(QUERY(x <* cartesian_point| SIZEOF(QUERY
       (y <* geometric_representation_context | item_in_context(x,y) AND
       (HIINDEX(x.coordinates) <> y.coordinate_space_dimension))) > 0 )) =0;

  -- ensure that the count of direction_ratios of each direction
  -- matches the coordinate_space_dimension of each geometric_context in
  -- which it is geometrically_founded
  WR2: SIZEOF(QUERY(x <* direction | SIZEOF( QUERY
       (y <* geometric_representation_context | item_in_context(x,y) AND
       (HIINDEX(x.direction_ratios) <> y.coordinate_space_dimension)))
       > 0 )) = 0;
END_RULE;
(*
```

Formal propositions:

**WR1:** There shall be no **cartesian_point** that has a number of coordinates that differs from the **coordinate_space_dimension** of the **geometric_representation_contexts** in which it is geometrically founded.

**WR2:** There shall be no **direction** that has a number of **direction_ratios** that differs from the **coordinate_space_dimension** of the **geometric_representation_contexts** in which it is geometrically founded.

NOTE – A check of only **cartesian_points** and **directions** is sufficient for all **geometric_representation_items** because:

a) All **geometric_representation_items** appear in trees of **representation_items** descending from the **items** attribute of entity **representation**. See WR1 of entity **representation_item** in ISO 10303-43.

b) Each **geometric_representation_item** gains its position and orientation information only by being, or referring to, a **cartesian_point** or **direction** entity in such a tree. In many cases this reference is made via an **axis_placement**.

c) No other use of any **geometric_representation_item** is allowed that would associate it with a coordinate space or otherwise assign a **dimension_count**.

## 4.6 geometry_schema function definitions

The EXPRESS language has a number of built-in functions. This section describes additional functions required for the definition and constraints on the geometry schema.

### 4.6.1 dimension_of

The function **dimension_of** returns the integer **dimension_count** of a **geometric_representation_context** in which the input **geometric_representation_item** is geometrically founded.

By virtue of the constraints in global rule **compatible_dimension**, this value is the **coordinate_space_dimension** of the input **geometric_representation_item**. See 4.5.1 for definition of this rule.

*EXPRESS* specification:

```
 *)
FUNCTION dimension_of(item : geometric_representation_item) :
  dimension_count;
  LOCAL
    x : SET OF representation;
    y : representation_context;
  END_LOCAL;

  -- Find the set of representation in which the item is used.

  x := using_representations(item);

  -- Determines the dimension_count of the
  -- geometric_representation_context. Note that the
  -- RULE compatible_dimension ensures that the context_of_items
  -- is of type geometric_representation_context and has
  -- the same dimension_count for all values of x.

    y := x[1].context_of_items;
    RETURN (y\geometric_representation_context.coordinate_space_dimension);

END_FUNCTION;
(*
```

Argument definitions:

**item:** (input) a **geometric_representation_item** for which the **dimension_count** is determined.

### 4.6.2 acyclic_curve_replica

The **acyclic_curve_replica** boolean function is a recursive function which determines whether, or not, a given **curve_replica** participates in its own definition. The function returns FALSE if the **curve_replica** refers to itself, directly or indirectly, in its own definition.

*EXPRESS* specification:

```
 *)
 FUNCTION acyclic_curve_replica(rep : curve_replica; parent : curve)
                                                      : BOOLEAN;
    IF NOT (('GEOMETRY_SCHEMA.CURVE_REPLICA') IN TYPEOF(parent)) THEN
       RETURN (TRUE);
    END_IF;
 (* Return TRUE if the parent is not of type curve_replica *)
    IF (parent :=: rep) THEN
       RETURN (FALSE);
  (* Return FALSE if the parent is the same curve_replica, otherwise,
    call function again with the parents own parent_curve.     *)
```

```
    ELSE RETURN(acyclic_curve_replica(rep, parent\curve_replica.parent_curve));
    END_IF;
  END_FUNCTION;
  (*
```

Argument definitions:

**rep:** (input) The **curve_replica** which is to be tested for a cyclic reference.

**parent:** (input) A **curve** used in the definition of the replica.

### 4.6.3 acyclic_point_replica

The **acyclic_point_replica** boolean function is a recursive function which determines whether, or not, a given **point_replica** participates in its own definition. The function returns FALSE if the **point_replica** refers to itself, directly or indirectly, in its own definition.

*EXPRESS* specification:

```
 *)
 FUNCTION acyclic_point_replica(rep : point_replica; parent : point)
                                                    : BOOLEAN;
   IF NOT (('GEOMETRY_SCHEMA.POINT_REPLICA') IN TYPEOF(parent)) THEN
      RETURN (TRUE);
   END_IF;
 (* Return TRUE if the parent is not of type point_replica *)
   IF (parent :=: rep) THEN
      RETURN (FALSE);
  (* Return FALSE if the parent is the same point_replica, otherwise,
   call function again with the parents own parent_pt.      *)
    ELSE RETURN(acyclic_point_replica(rep, parent\point_replica.parent_pt));
    END_IF;
  END_FUNCTION;
  (*
```

Argument definitions:

**rep:** (input) The **point_replica** which is to be tested for a cyclic reference.

**parent:** (input) A **point** used in the definition of the replica.



### 4.6.4 acyclic_surface_replica

The **acyclic_surface_replica** boolean function is a recursive function which determines whether, or not, a given **surface_replica** participates in its own definition. The function returns FALSE if the **surface_replica** refers to itself, directly or indirectly, in its own definition.

*EXPRESS* specification:

```
 *)
 FUNCTION acyclic_surface_replica(rep : surface_replica; parent : surface)
                                                    : BOOLEAN;
```

```
  IF NOT (('GEOMETRY_SCHEMA.SURFACE_REPLICA') IN TYPEOF(parent)) THEN
     RETURN (TRUE);
  END_IF;
(* Return TRUE if the parent is not of type surface_replica *)
  IF (parent :=: rep) THEN
     RETURN (FALSE);
 (* Return FALSE if the parent is the same surface_replica, otherwise,
  call function again with the parents own parent_surface.     *)
   ELSE RETURN(acyclic_surface_replica(rep,
                 parent\surface_replica.parent_surface));
   END_IF;
 END_FUNCTION;
 (*
```

Argument definitions:

**rep:** (input) The **surface_replica** which is to be tested for a cyclic reference.

**parent:** (input) A **surface** used in the definition of the replica.

### 4.6.5 associated_surface

This function determines the unique surface which is associated with the **pcurve_or_surface** type. It is required by the propositions which apply to surface curve and its subtypes.

*EXPRESS* specification:

```
*)
FUNCTION associated_surface(arg : pcurve_or_surface) : surface;
  LOCAL
    surf : surface;
  END_LOCAL;

  IF 'GEOMETRY_SCHEMA.PCURVE' IN TYPEOF(arg) THEN
    surf := arg.basis_surface;
  ELSE
    surf := arg;
  END_IF;
  RETURN(surf);
END_FUNCTION;
(*
```

Argument definitions:

**arg:** (input) The pcurve or surface for which the determination of the associated parent surface is required.

**surf:** (output) The parent surface associated with **arg**.

### 4.6.6 base_axis

This function returns three normalised orthogonal directions, u[1], u[2] and u[3]. In the three-dimensional case, with complete input data, u[3] is in the direction of axis3, u[1] is in the direction of the projection of axis1 onto the plane normal to u[3], and u[2] is orthogonal to both u[1] and u[3], taking the same sense as axis2. In the two-dimensional case u[1] is in the direction of axis1 and u[2] is perpendicular to this, taking its sense from axis2. For incomplete input data suitable default values are derived.

*EXPRESS* specification:

```
*)
FUNCTION base_axis(dim : INTEGER; axis1, axis2, axis3 : direction) :
                                                 LIST [2:3] OF direction;
  LOCAL
    vec    : direction;
    u      : LIST [2:3] OF direction;
    factor : REAL;
  END_LOCAL;

  IF (dim = 3) THEN
    u[3] := NVL(normalise(axis3), direction([0.0,0.0,1.0]));
    u[1] := first_proj_axis(u[3],axis1);
    u[2] := second_proj_axis(u[3],u[1],axis2);
  ELSE
    u[3] := ?;
    IF EXISTS(axis1) THEN
      u[1] := normalise(axis1);
      u[2] := orthogonal_complement(u[1]);
      IF EXISTS(axis2) THEN
        factor := dot_product(axis2,u[2]);
        IF (factor < 0.0) THEN
          u[2].direction_ratios[1] := -u[2].direction_ratios[1];
          u[2].direction_ratios[2] := -u[2].direction_ratios[2];
        END_IF;
      END_IF;
    ELSE
      IF EXISTS(axis2) THEN
        u[2] := normalise(axis2);
        u[1] := orthogonal_complement(u[2]);
        u[1].direction_ratios[1] := -u[1].direction_ratios[1];
        u[1].direction_ratios[2] := - u[1].direction_ratios[2];
      ELSE
        u[1].direction_ratios[1] := 1.0;
        u[1].direction_ratios[2] := 0.0;
        u[2].direction_ratios[1] := 0.0;
        u[2].direction_ratios[2] := 1.0;
      END_IF;
    END_IF;
  END_IF;
  RETURN(u);
END_FUNCTION;
(*
```

Argument definitions:

**dim:** (input) The integer value of the dimensionality of the space in which the normalised orthogonal directions are required.

**axis1:** (input) A direction used as a first approximation to the direction of output axis u[1].

**axis2:** (input) A direction used to determine the sense of u[2].

**axis3:** (input) The direction of u[3] in the case dim=3, or NULL in the case dim=2.

**u:** (output) A list of dim (i.e., 2 or 3) mutually perpendicular directions.

### 4.6.7 build_2axes

This function returns two normalised orthogonal directions. **u[1]** is in the direction of **ref_direction** and **u[2]** is perpendicular to **u[1]**. A default value of (1.0,0.0) is supplied for **ref_direction** if the input data is incomplete.

*EXPRESS* specification:

```
*)
FUNCTION build_2axes(ref_direction : direction) : LIST [2:2] OF direction;
  LOCAL
    u : LIST[2:2] OF direction;
  END_LOCAL;

  u[1] := NVL(normalise(ref_direction), direction([1.0,0.0]));
  u[2] := orthogonal_complement(u[1]);
  RETURN(u);
END_FUNCTION;
(*
```

Argument definitions:

**ref_direction:** (input) A reference direction in 2 dimensional space, this may be defaulted to [1.0,0.0].

**u:** (output) A list of 2 mutually perpendicular directions, u[1] is parallel to ref_direction.

### 4.6.8 build_axes

This function returns three normalised orthogonal directions. u[3] is in the direction of **axis**, u[1] is in the direction of the projection of **ref_direction** onto the plane normal to u[3] and u[2] is the cross product of u[3] and u[1]. Default values are supplied if input data is incomplete.

*EXPRESS* specification:

```
*)
FUNCTION build_axes(axis, ref_direction : direction) :
```

```
                                        LIST [3:3] OF direction;
  LOCAL
    u : LIST[3:3] OF direction;
  END_LOCAL;

  u[3] := NVL(normalise(axis), direction([0.0,0.0,1.0]));
  u[1] := first_proj_axis(u[3],ref_direction);
  u[2] := normalise(cross_product(u[3],u[1])).orientation;
  RETURN(u);
END_FUNCTION;
(*
```

Argument definitions:

**axis:** (input) The intended direction of u[3], this may be defaulted to [0.0,0.0,1.0].

**ref_direction:** (input) A direction in a direction used to compute u[1].

**u:** (output) A list of 3 mutually orthogonal directions in 3D space.

### 4.6.9 orthogonal_complement

This function returns a direction which is the orthogonal complement of the input direction. The input direction must be a two-dimensional direction and the result is a vector of the same type and perpendicular to the input vector.

*EXPRESS* specification:

```
*)
FUNCTION orthogonal_complement(vec : direction) : direction;
  LOCAL
    result : direction;
  END_LOCAL;

  IF (vec.dim <> 2) OR NOT EXISTS (vec) THEN
    RETURN(?);
  ELSE
    result.direction_ratios[1] := -vec.direction_ratios[2];
    result.direction_ratios[2] := vec.direction_ratios[1];
    RETURN(result);
  END_IF;
END_FUNCTION;
(*
```

Argument definitions:

**vec:** (input) A direction in 2D space.

**result:** (output) A direction orthogonal to vec.

## 4.6.10 first_proj_axis

This function produces a 3-dimensional direction which is, with fully defined input, the projection of **arg** onto the plane normal to the **z_axis**. With **arg** defaulted the result is the projection of [1,0,0] onto this plane; except that if **z_axis** = [1,0,0], [0,1,0] is the default for **arg**. A violation occurs if **arg** is in the same direction as the input **z_axis**.

*EXPRESS* specification:

```
*)
FUNCTION first_proj_axis(z_axis, arg : direction) : direction;
  LOCAL
    x_axis : direction;
    v      : direction;
    z      : direction;
    x_vec  : vector;
  END_LOCAL;

  IF (NOT EXISTS(z_axis)) OR (NOT EXISTS(arg)) OR (arg.dim <> 3) THEN
    x_axis := ?;
  ELSE
    z_axis := normalise(z_axis);
    IF NOT EXISTS(arg) THEN
      IF (z_axis <> direction([1.0,0.0,0.0])) THEN
        v := direction([1.0,0.0,0.0]);
      ELSE
        v := direction([0.0,1.0,0.0]);
      END_IF;
    ELSE
      IF ((cross_product(arg,z).magnitude) = 0.0) THEN
        RETURN (?);
      ELSE
        v := normalise(arg);
      END_IF;
    END_IF;
    x_vec := scalar_times_vector(dot_product(v, z), z_axis);
    x_axis := vector_difference(v, x_vec).orientation;
    x_axis := normalise(x_axis);
  END_IF;
  RETURN(x_axis);
END_FUNCTION;
(*
```



Argument definitions:

**z_axis:** (input) A direction defining a local Z axis.

**arg:** (input) A direction not parallel to z_axis.

**x_axis:** (output) A direction which is in the direction of the projection of arg onto the plane with normal z_axis.

### 4.6.11 second_proj_axis

This function returns the normalised vector that is simultaneously the projection of **arg** onto the plane normal to the vector **z_axis** and onto the plane normal to the vector **x_axis**. If **arg** is NULL, the projection of the vector (0,1,0) onto **z_axis** is returned.

*EXPRESS* specification:

```
*)
FUNCTION second_proj_axis(z_axis, x_axis, arg: direction) : direction;
  LOCAL
    y_axis : vector;
    v      : direction;
    temp   : vector;
  END_LOCAL;

  IF NOT EXISTS(arg) THEN
    v := direction([0.0,1.0,0.0]);
  ELSE
    v := arg;
  END_IF;

  temp   := scalar_times_vector(dot_product(v, z_axis), z_axis);
  y_axis := vector_difference(v, temp);
  temp   := scalar_times_vector(dot_product(v, x_axis), x_axis);
  y_axis := vector_difference(y_axis, temp);
  y_axis := normalise(y_axis);
  RETURN(y_axis.orientation);
END_FUNCTION;
(*
```

Argument definitions:

**z_axis:** (input) A direction defining a local Z axis.

**x_axis:** (input) A direction not parallel to **z_axis**.

**arg:** (input) A direction which is used as the first approximation to the direction of **y_axis**.

**y_axis.orientation:** (output) A direction determined by first projecting arg onto the plane with normal **z_axis**, then projecting the result onto the plane normal to **x_axis**.



### 4.6.12 cross_product

This function returns the vector, or cross, product of two input directions. The input directions must be three-dimensional and are normalised at the start of the computation. The result is always a vector which is unitless. If the input directions are either parallel or anti-parallel, a vector of zero magnitude is returned with **orientation** = **arg1**.

*EXPRESS* specification:

```
*)
FUNCTION cross_product (arg1, arg2 : direction) : vector;
  LOCAL
    mag    : REAL;
    res    : direction;
    v1,v2  : LIST[3:3] OF REAL;
    result : vector;
  END_LOCAL;

  IF ( NOT EXISTS (arg1) OR (arg1.dim = 2)) OR
     ( NOT EXISTS (arg2) OR (arg2.dim = 2)) THEN
    RETURN(?);
  ELSE
    BEGIN
      v1  := normalise(arg1).direction_ratios;
      v2  := normalise(arg2).direction_ratios;
      res.direction_ratios[1]  := (v1[2]*v2[3] - v1[3]*v2[2]);
      res.direction_ratios[2]  := (v1[3]*v2[1] - v1[1]*v2[3]);
      res.direction_ratios[3]  := (v1[1]*v2[2] - v1[2]*v2[1]);
      mag := 0.0;
      REPEAT i := 1 TO 3;
        mag := mag + res.direction_ratios[i]*res.direction_ratios[i];
      END_REPEAT;
      IF (mag > 0.0) THEN
        result.orientation := res;
        result.magnitude := SQRT(mag);
      ELSE
        result.orientation := arg1;
        result.magnitude := 0.0;
      END_IF;
      RETURN(result);
    END;
  END_IF;
END_FUNCTION;
(*
```

Argument definitions:

**arg1:** (input) A direction defining first vector in cross product operation.

**arg2:** (input) A direction defining second operand for cross product.

**result:** (output) A vector which is the cross product of **arg1** and **arg2**.



## 4.6.13 dot_product

This function returns the scalar, or dot (·), product of two directions. The input arguments can be directions in either two- or three-dimensional space and are normalised at the start of the computation. The returned scalar is undefined if the input directions have different dimensionality, or if either is undefined.

*EXPRESS* specification:

```
*)
FUNCTION dot_product(arg1, arg2 : direction) : REAL;
  LOCAL
    scalar : REAL;
    vec1, vec2: direction;
    ndim : INTEGER;
  END_LOCAL;

  IF NOT EXISTS (arg1) OR NOT EXISTS (arg2) THEN
    scalar := ?;
    (* When function is called with invalid data a NULL result is returned *)
  ELSE
    IF (arg1.dim <> arg2.dim) THEN
      scalar := ?;
    (* When function is called with invalid data a NULL result is returned *)
    ELSE
      BEGIN
        vec1   := normalise(arg1);
        vec2   := normalise(arg2);
        ndim   := arg1.dim;
        scalar := 0.0;
        REPEAT  i := 1 TO ndim;
          scalar := scalar +
                      vec1.direction_ratios[i]*vec2.direction_ratios[i];
        END_REPEAT;
      END;
    END_IF;
  END_IF;
  RETURN (scalar);
END_FUNCTION;
(*
```

Argument definitions:

**arg1:** (input) A direction defining first vector in dot product, or scalar product operation.

**arg2:** (input) A direction defining second operand for dot product.

**result:** (output) A scalar which is the dot product of **arg1** and **arg2**.

### 4.6.14 normalise

This function returns a vector or direction whose components are normalised to have a sum of squares of 1.0. The output is of the same type (**direction** or **vector**, with the same units) as the input argument. If the input argument is not defined or is of zero length, the output vector is undefined.

*EXPRESS* specification:

```
*)
```

```
FUNCTION normalise (arg : vector_or_direction) : vector_or_direction;
  LOCAL
    ndim   : INTEGER;
    v      : direction;
    result : vector_or_direction;
    vec    : vector;
    mag    : REAL;
  END_LOCAL;

  IF NOT EXISTS (arg) THEN
    result := ?;
    (* When function is called with invalid data a NULL result is returned *)
  ELSE
    ndim := arg.dim;
    IF 'GEOMETRY_SCHEMA.VECTOR' IN TYPEOF(arg) THEN
      BEGIN
        vec := arg;
        v := arg.orientation;

        IF arg.magnitude = 0.0 THEN
          RETURN(?);
        ELSE
          vec.magnitude := 1.0;
        END_IF;
      END;
    ELSE
      v := arg;
    END_IF;
    mag := 0.0;
    REPEAT  i := 1 TO ndim;
      mag := mag + v.direction_ratios[i]*v.direction_ratios[i];
    END_REPEAT;
    IF mag > 0.0 THEN
      mag := SQRT(mag);
      REPEAT  i := 1 TO ndim;
        v.direction_ratios[i] := v.direction_ratios[i]/mag;
      END_REPEAT;
      IF 'GEOMETRY_SCHEMA.VECTOR' IN TYPEOF(arg) THEN
        vec.orientation := v;
        result := vec;
      ELSE
        result := v;
      END_IF;
    ELSE
      RETURN(?);
    END_IF;
  END_IF;
  RETURN (result);
END_FUNCTION;
(*
```

Argument definitions:

**arg:** (input) A vector or direction to be normalised.

**result:** (output) A vector or direction which is parallel to **arg1** and of unit length.

### 4.6.15 scalar_times_vector

This function returns the vector that is the scalar multiple of the input vector. It accepts as input a scalar and a 'vector' which may be either a **direction** or a **vector**. The output is a **vector** of the same units as the input vector, or unitless if a direction is input. If either input argument is undefined, the returned vector is also undefined. The **orientation** of the **vector** is reversed if the scalar is negative.

*EXPRESS* specification:

```
 *)
 FUNCTION scalar_times_vector (scalar : REAL; vec : vector_or_direction)
                                      : vector;
   LOCAL
     v      : direction;
     mag    : REAL;
     result : vector;
   END_LOCAL;

   IF NOT EXISTS (scalar) OR NOT EXISTS (vec) THEN
     result := ?;
   (* When function is called with invalid data a NULL result is returned *)
   ELSE
     IF 'GEOMETRY_SCHEMA.VECTOR' IN TYPEOF (vec) THEN
       v   := vec.orientation;
       mag := scalar * vec.magnitude;
     ELSE
       v   := vec;
       mag := scalar;
     END_IF;
     IF (mag < 0.0 ) THEN
       REPEAT i := 1 TO SIZEOF(v.direction_ratios);
         v.direction_ratios[i] := -v.direction_ratios[i];
       END_REPEAT;
       mag := -mag;
     END_IF;
     result.orientation := normalise(v);
     result.magnitude   := mag;
   END_IF;
   RETURN (result);
END_FUNCTION;
 (*
```

Argument definitions:

**scalar:** (input) A real number to participate in the product.

**vec:** (input) A vector or direction which is to be multiplied.

**result:** (output) A vector which is the product of **scalar** and **vec**.

## 4.6.16 vector_sum

This function returns the sum of the input arguments. The function returns as a vector the vector sum of the two input 'vectors'. For this purpose **directions** are treated as unit vectors. The input arguments must both be of the same dimensionality but may be either directions or vectors. Where both arguments are vectors, they must be expressed in the same units. A zero sum vector produces a vector of zero magnitude in the direction of **arg1**. If both input arguments are directions, the result is unitless.

*EXPRESS* specification:

```
*)
FUNCTION vector_sum(arg1, arg2 : vector_or_direction) : vector;
  LOCAL
    result           : vector;
    res, vec1, vec2  : direction;
    mag, mag1, mag2  : REAL;
    ndim             : INTEGER;
  END_LOCAL;

  IF ((NOT EXISTS (arg1)) OR (NOT EXISTS (arg2))) OR (arg1.dim <> arg2.dim)
      THEN
    result := ?;
  (* When function is called with invalid data a NULL result is returned *)
  ELSE
    BEGIN
      IF 'GEOMETRY_SCHEMA.VECTOR' IN TYPEOF(arg1) THEN
        mag1 := arg1.magnitude;
        vec1 := arg1.orientation;
      ELSE
        mag1 := 1.0;
        vec1 := arg1;
      END_IF;
      IF 'GEOMETRY_SCHEMA.VECTOR' IN TYPEOF(arg2) THEN
        mag2 := arg2.magnitude;
        vec2 := arg2.orientation;
      ELSE
        mag2 := 1.0;
        vec2 := arg2;
      END_IF;
      vec1 := normalise (vec1);
      vec2 := normalise (vec2);
      ndim := SIZEOF(vec1.direction_ratios);
      mag := 0.0;
      REPEAT i := 1 TO ndim;
        res.direction_ratios[i] := mag1*vec1.direction_ratios[i] +
                                   mag2*vec2.direction_ratios[i];
        mag := mag + (res.direction_ratios[i]*res.direction_ratios[i]);
      END_REPEAT;
      IF (mag > 0.0 ) THEN
```

```
        result.magnitude := SQRT(mag);
        result.orientation := res;
      ELSE
        result.magnitude := 0.0;
        result.orientation := vec1;
      END_IF;
    END;
  END_IF;
  RETURN (result);
END_FUNCTION;
(*
```

Argument definitions:

**arg1:** (input) A direction defining first vector in vector sum operation.

**arg2:** (input) A direction defining second operand for vector sum.

**result:** (output) A vector which is the vector sum of **arg1** and **arg2**.

## 4.6.17 vector_difference

This function returns the difference of the input arguments as (**arg1** - **arg2**). The function returns as a vector the vector difference of the two input 'vectors'. For this purpose **directions** are treated as unit vectors. The input arguments shall both be of the same dimensionality but may be either directions or vectors. If both input arguments are vectors, they must be expressed in the same units; if both are directions, a unitless result is produced. A zero difference vector produces a vector of zero magnitude in the direction of **arg1**.

*EXPRESS* specification:

```
*)
FUNCTION vector_difference(arg1, arg2 : vector_or_direction) : vector;
  LOCAL
    result          : vector;
    res, vec1, vec2 : direction;
    mag, mag1, mag2 : REAL;
    ndim            : INTEGER;
  END_LOCAL;

  IF ((NOT EXISTS (arg1)) OR (NOT EXISTS (arg2))) OR (arg1.dim <> arg2.dim)
      THEN
    result := ?;
  (* When function is called with invalid data a NULL result is returned *)
  ELSE
    BEGIN
      IF 'GEOMETRY_SCHEMA.VECTOR' IN TYPEOF(arg1) THEN
        mag1 := arg1.magnitude;
        vec1 := arg1.orientation;
      ELSE
        mag1 := 1.0;
```

```
      vec1 := arg1;
    END_IF;
    IF 'GEOMETRY_SCHEMA.VECTOR' IN TYPEOF(arg2) THEN
      mag2 := arg2.magnitude;
      vec2 := arg2.orientation;
    ELSE
      mag2 := 1.0;
      vec2 := arg2;
    END_IF;
    vec1 := normalise (vec1);
    vec2 := normalise (vec2);
    ndim := SIZEOF(vec1.direction_ratios);
    mag := 0.0;
    REPEAT i := 1 TO ndim;
      res.direction_ratios[i] := mag1*vec1.direction_ratios[i] -
                                       mag2*vec2.direction_ratios[i];
      mag := mag + (res.direction_ratios[i]*res.direction_ratios[i]);
    END_REPEAT;
    IF (mag > 0.0 ) THEN
      result.magnitude := SQRT(mag);
      result.orientation := res;
    ELSE
      result.magnitude := 0.0;
      result.orientation := vec1;
    END_IF;
  END;
 END_IF;
 RETURN (result);
END_FUNCTION;
(*
```

Argument definitions:

**arg1:** (input) A direction defining first vector in vector difference operation.

**arg2:** (input) A direction defining second operand for vector difference.

**result:** (output) A vector which is the vector difference of **arg1** and **arg2**.

## 4.6.18 default_b_spline_knot_mult

This function returns the INTEGER array of knot multiplicities, depending on the type of knot vector, for the B-spline parametrisation.

*EXPRESS* specification:

```
*)
FUNCTION default_b_spline_knot_mult(degree, up_knots : INTEGER;
                                    uniform : knot_type)
                                            : LIST [2:?] OF INTEGER;
  LOCAL
    knot_mult : LIST [1:up_knots] OF INTEGER;
```

```
  END_LOCAL;

  REPEAT i := 1 TO up_knots;
    knot_mult[i] := 0;
  END_REPEAT;

  IF uniform = uniform_knots THEN
    REPEAT i := 1 TO up_knots;
      knot_mult[i] := 1;
    END_REPEAT;
  END_IF;

  IF uniform = quasi_uniform_knots THEN
    knot_mult[1] := degree + 1;
    knot_mult[up_knots] := degree + 1;

    REPEAT i := 2 TO (up_knots - 1);
      knot_mult[i] := 1;
    END_REPEAT;
  END_IF;

  IF uniform = piecewise_bezier_knots THEN
    knot_mult[1] := degree +1;
    knot_mult[up_knots] := degree + 1;

    REPEAT i := 2 TO (up_knots - 1);
      knot_mult[i] := degree;
    END_REPEAT;
  END_IF;
  RETURN(knot_mult);
END_FUNCTION;
(*
```

Argument definitions:

**degree:** (input) An integer defining the degree of the B-spline basis functions.

**up_knots:** (input) An integer which gives the number of knot multiplicities required.

**uniform:** (input) The type of basis function for which knot multiplicities are required.

**knot_mult:** (output) A list of integer knot multiplicities.

### 4.6.19 default_b_spline_knots

This function returns the knot vector, depending on the **knot_type**, for a B-spline parametrisation.

*EXPRESS* specification:

```
*)
FUNCTION default_b_spline_knots(degree,up_knots : INTEGER;
```

```
                                    uniform : knot_type)
                                            : LIST [2:?] OF parameter_value;
  LOCAL
    knots  : LIST [1:up_knots] OF parameter_value;
    ishift : INTEGER := 1;
  END_LOCAL;

  REPEAT i := 1 TO up_knots;
    knots[i] := 0;
  END_REPEAT;
  IF (uniform = uniform_knots) THEN
     ishift := degree + 1;
  END_if;
  IF (uniform = uniform_knots) OR
     (uniform = quasi_uniform_knots) OR
     (uniform = piecewise_bezier_knots) THEN

    REPEAT i := 1 TO up_knots;
      knots[i] := i - ishift;
    END_REPEAT;
  END_IF;
  RETURN(knots);
 END_FUNCTION;
 (*
```

Argument definitions:

**degree:** (input) An integer defining the degree of the B-spline basis functions.

**up_knots:** (input) An integer which gives the number of knot values required.

**uniform:** (input) The type of basis function for which knots are required.

**knots:** (output) A list of parameter values for the knots.

## 4.6.20 default_b_spline_curve_weights

This function returns **up_cp** weights equal to 1.0 in an ARRAY OF REAL.

*EXPRESS* specification:

```
 *)
 FUNCTION default_b_spline_curve_weights(up_cp : INTEGER)
                                          : ARRAY [0:up_cp] OF REAL;
   LOCAL
     weights : ARRAY [0:up_cp] OF REAL;
   END_LOCAL;

   REPEAT i := 0 TO up_cp;
     weights[i] := 1;
   END_REPEAT;
   RETURN(weights);
```

```
END_FUNCTION;
(*
```

Argument definitions:

**up_cp:** (input) An integer defining the upper index on the array of the B-spline curve weights required.

**weights:** (output) A real array of weight values.

NOTE – This function is not used in this part of ISO 10303 but is defined here for use by applications.

### 4.6.21 default_b_spline_surface_weights

This function returns weights equal to 1.0 in an ARRAY OF ARRAY OF REAL.

*EXPRESS* specification:

```
*)
FUNCTION default_b_spline_surface_weights(u_upper, v_upper: INTEGER)
                                     : ARRAY [0:u_upper] OF
                                       ARRAY [0:v_upper] OF REAL;
  LOCAL
    weights : ARRAY [0:u_upper] OF ARRAY [0:v_upper] OF REAL;
  END_LOCAL;

  REPEAT i := 0 TO u_upper;
    REPEAT j := 0 TO v_upper;
      weights[i][j] := 1;
    END_REPEAT;
  END_REPEAT;
  RETURN(weights);
END_FUNCTION;
(*
```

Argument definitions:

**u_upper:** (input) An integer defining the upper index on the array of the B-spline surface weights required in the u direction.

**v_upper:** (input) An integer giving the upper index of the number of weights required for the surface in the v parameter direction.

**weights:** (output) A real array of array of weight values.

NOTE – This function is not used in this part of ISO 10303 but is defined here for use by applications.

### 4.6.22 constraints_param_b_spline

This function checks the parametrisation of a B-spline curve or (one of the directions of) a B-spline surface and returns TRUE if no inconsistencies are found.

These constraints are:

a) Degree $\geq 1$.

b) Upper index on knots $\geq 2$.

c) Upper index on control points $\geq$ degree.

d) Sum of knot multiplicities = degree + (upper index on control points) + 2.

e) For the first and last knot the multiplicity is bounded by 1 and (degree+1).

f) For all other knots the knot multiplicity is bounded by 1 and degree.

g) The consecutive knots are increasing in value.

*EXPRESS* specification:

```
*)
FUNCTION constraints_param_b_spline(degree, up_knots, up_cp : INTEGER;
                                   knot_mult : LIST OF INTEGER;
                      knots : LIST OF parameter_value) : BOOLEAN;
  LOCAL
    result   : BOOLEAN := TRUE;
    k,l,sum : INTEGER;
  END_LOCAL;

  (* Find sum of knot multiplicities. *)
  sum := knot_mult[1];

  REPEAT i := 2 TO up_knots;
    sum := sum + knot_mult[i];
  END_REPEAT;

  (* Check limits holding for all B-spline parametrisations *)
  IF (degree < 1) OR (up_knots < 2) OR (up_cp < degree) OR
       (sum <> (degree + up_cp + 2)) THEN
    result := FALSE;
    RETURN(result);
  END_IF;

  k := knot_mult[1];

  IF (k < 1) OR (k > degree + 1) THEN
    result := FALSE;
    RETURN(result);
  END_IF;
```

```
  REPEAT i := 2 TO up_knots;
    IF (knot_mult[i] < 1) OR (knots[i] <= knots[i-1]) THEN
      result := FALSE;
      RETURN(result);
    END_IF;

    k := knot_mult[i];

    IF (i < up_knots) AND (k > degree) THEN
      result := FALSE;
      RETURN(result);
    END_IF;

    IF (i = up_knots) AND (k > degree + 1) THEN
      result := FALSE;
      RETURN(result);
    END_IF;
  END_REPEAT;
  RETURN(result);
END_FUNCTION;
(*
```

Argument definitions:

**degree:** (input) An integer defining the degree of the B-spline basis functions.

**up_knots:** (input) An integer giving the upper index of the list of knot multiplicities.

**up_cp:** (input) An integer which is the upper index of the control points for the curve or surface being checked for consistency of its parameter values.

**knot_mult:** (input) The list of knot multiplicities.

### 4.6.23 curve_weights_positive

This function checks the weights associated with the control points of a **rational_b_spline_curve** and returns TRUE if they are all positive.

*EXPRESS* specification:

```
*)
FUNCTION curve_weights_positive(b: rational_b_spline_curve) : BOOLEAN;
  LOCAL
    result : BOOLEAN := TRUE;
  END_LOCAL;

  REPEAT i := 0 TO b.upper_index_on_control_points;
    IF b.weights[i] <= 0.0  THEN
      result := FALSE;
      RETURN(result);
    END_IF;
  END_REPEAT;
```

```
  RETURN(result);
 END_FUNCTION;
(*
```

Argument definitions:

**b:** (input) A rational B-spline curve for which the weight values are to be tested.

### 4.6.24 constraints_composite_curve_on_surface

This function checks that the curves referenced by the segments of the **composite_curve_on_surface** are all curves on surface, including the **composite_curve_on_surface** type, which is admissible as a **bounded_curve**.

*EXPRESS* specification:

```
*)
FUNCTION constraints_composite_curve_on_surface
              (c: composite_curve_on_surface) : BOOLEAN;
  LOCAL
    n_segments : INTEGER := SIZEOF(c.segments);
  END_LOCAL;

  REPEAT k := 1 TO n_segments;
    IF (NOT('GEOMETRY_SCHEMA.PCURVE' IN
          TYPEOF(c\composite_curve.segments[k].parent_curve))) AND
       (NOT('GEOMETRY_SCHEMA.SURFACE_CURVE' IN
          TYPEOF(c\composite_curve.segments[k].parent_curve))) AND
       (NOT('GEOMETRY_SCHEMA.COMPOSITE_CURVE_ON_SURFACE' IN
          TYPEOF(c\composite_curve.segments[k].parent_curve)))  THEN
      RETURN (FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
(*
```

Argument definitions:

**c:** (input) A composite curve on surface to be verified.



### 4.6.25 get_basis_surface

This function returns the basis surface for a curve as a a SET of **surfaces**. For a curve which is not a **curve_on_surface** an empty SET is returned.

*EXPRESS* specification:

```
*)
FUNCTION get_basis_surface (c : curve_on_surface) : SET[0:2] OF surface;
  LOCAL
    surfs  : SET[0:2] OF surface;
```

```
    n       : INTEGER;
  END_LOCAL;
  surfs := [];
  IF 'GEOMETRY_SCHEMA.PCURVE' IN TYPEOF (c) THEN
    surfs := [c\pcurve.basis_surface];
  ELSE
    IF 'GEOMETRY_SCHEMA.SURFACE_CURVE' IN TYPEOF (c) THEN
      n := SIZEOF(c\surface_curve.associated_geometry);
      REPEAT i := 1 TO n;
      surfs := surfs +
                associated_surface(c\surface_curve.associated_geometry[i]);
      END_REPEAT;
    END_IF;
  END_IF;
  IF 'GEOMETRY_SCHEMA.COMPOSITE_CURVE_ON_SURFACE' IN TYPEOF (c) THEN
   (* For a composite_curve_on_surface the basis_surface is the intersection
    of the basis_surfaces of all the segments. *)
     n := SIZEOF(c\composite_curve_on_surface.segments);
     surfs := get_basis_surface(c\composite_curve_on_surface.segments[1].parent_curve);
     IF n > 1 THEN
       REPEAT i := 2 TO n;
 surfs := surfs * get_basis_surface(c\composite_curve_on_surface.segments[i].parent_curve);
       END_REPEAT;
     END_IF;
  END_IF;
  RETURN(surfs);
END_FUNCTION;
(*
```

<u>Argument definitions:</u>

**c:** (input) A curve for which the **basis_surface** is to be determined.

**surf:** (output) The set containing the **basis_surface** or surfaces on which **c** lies.

### 4.6.26 surface_weights_positive

This function checks the weights associated with the control points of a **rational_b_spline_surface** and returns TRUE if they are all positive.

<u>*EXPRESS* specification:</u>

```
*)
FUNCTION surface_weights_positive(b: rational_b_spline_surface) : BOOLEAN;
  LOCAL
    result          : BOOLEAN := TRUE;
  END_LOCAL;

  REPEAT i := 0 TO b.u_upper;
    REPEAT j := 0 TO b.v_upper;
      IF (b.weights[i][j] <= 0.0)  THEN
        result := FALSE;
        RETURN(result);
```

```
      END_IF;
    END_REPEAT;
  END_REPEAT;
  RETURN(result);
END_FUNCTION;
(*
```

Argument definitions:

**b:** (input) A rational B-spline surface for which the weight values are to be tested.

### 4.6.27 constraints_rectangular_composite_surface

This functions checks the following constraints on the attributes of a rectangular composite surface:

- that the component surfaces are all either rectangular trimmed surfaces or B-spline surfaces;
- that the **transition** attributes of the segments array do not contain the value **discontinuous** except for the last row or column, where they indicate that the surface is not closed in the appropriate direction.

*EXPRESS* specification:

```
*)
FUNCTION constraints_rectangular_composite_surface
        (s : rectangular_composite_surface) : BOOLEAN;

(* Check the surface types *)
  REPEAT i := 1 TO s.n_u;
    REPEAT j := 1 TO s.n_v;
      IF NOT (('GEOMETRY_SCHEMA.B_SPLINE_SURFACE' IN TYPEOF
                (s.segments[i][j].parent_surface)) OR
             ('GEOMETRY_SCHEMA.RECTANGULAR_TRIMMED_SURFACE' IN TYPEOF
                (s.segments[i][j].parent_surface))) THEN
        RETURN(FALSE);
      END_IF;
    END_REPEAT;
  END_REPEAT;

  (* Check the transition codes, omitting the last row or column *)
  REPEAT i := 1 TO s.n_u-1;
    REPEAT j := 1 TO s.n_v;
      IF s.segments[i][j].u_transition = discontinuous THEN
        RETURN(FALSE);
      END_IF;
    END_REPEAT;
  END_REPEAT;

  REPEAT i := 1 TO s.n_u;
    REPEAT j := 1 TO s.n_v-1;
```

```
      IF s.segments[i][j].v_transition = discontinuous THEN
        RETURN(FALSE);
      END_IF;
    END_REPEAT;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;
(*
```

Argument definitions:

**s:** (input) A rectangular composite surface to be verified.

### 4.6.28 list_to_array

The function **list_to_array** converts a generic list to an array with pre-determined array bounds. If the array bounds are incompatible with the number of elements in the original list, a null result is returned. This function is used to construct the arrays of control points and weights used in the b-spline entities.

*EXPRESS* specification:

```
*)
FUNCTION list_to_array(lis : LIST [0:?] OF GENERIC : T;
                     low,u : INTEGER) : ARRAY[low:u] OF GENERIC : T;
  LOCAL
    n   : INTEGER;
    res : ARRAY [low:u] OF GENERIC : T;
  END_LOCAL;

  n := SIZEOF(lis);
  IF (n <> (u-low +1)) THEN
    RETURN(?);
  ELSE
    REPEAT i := 1 TO n;
      res[low+i-1] := lis[i];
    END_REPEAT;
    RETURN(res);
  END_IF;
END_FUNCTION;
(*
```

Argument definitions:

**lis:** (input) A list to be converted.

**low:** (input) An integer specifying the required lower index of the output array.

**u:** (input) An integer value for the upper index.

**res:** (output) The array generated from the input data.

### 4.6.29 make_array_of_array

The function **make_array_of_array** builds an array of arrays from a list of lists. The function first checks that the specified array dimensions are compatible with the sizes of the lists, and in particular, verifies that all the sub-lists contain the same number of elements. A null result is returned if the input data is incompatible with the dimensions. This function is used to construct the arrays of control points and weights for a B-spline surface.

*EXPRESS* specification:

```
*)
FUNCTION make_array_of_array(lis : LIST[1:?] OF LIST [1:?] OF GENERIC : T;
                             low1, u1, low2, u2 : INTEGER):
                 ARRAY[low1:u1] OF ARRAY [low2:u2] OF GENERIC : T;
  LOCAL
    n1,n2 : INTEGER;
    res   : ARRAY[low1:u1] OF ARRAY [low2:u2] OF GENERIC : T;
    resl  : LIST[1:?] OF ARRAY [low2:u2] OF GENERIC : T;
  END_LOCAL;

  (* Check input dimensions for consistency *)
  n1 := SIZEOF(lis);
  n2 := SIZEOF(lis[1]);

  IF (n1 <> (u1 -low1 + 1)) AND (n2 <> (u2 - low2 + 1)) THEN
    RETURN(?);
  END_IF;

  REPEAT i := 1 TO n1;
    IF (SIZEOF(lis[i]) <> n2) THEN
      RETURN(?);
    END_IF;
  END_REPEAT;

  (* Build a list of sub-arrays *)
  REPEAT i := 1 TO  n1;
    RESL[i] := list_to_array(lis[i],low2,u2);
  END_REPEAT;

  res := list_to_array(resl,low1,u1);
  RETURN(res);
END_FUNCTION;
(*
```



Argument definitions:

**lis:** (input) A list of list to be converted.

**low1:** (input) An integer specifying the required lower index of the first output array.

**u1:** (input) An integer value for the upper index of the first output array.

**low2:** (input) An integer specifying the required lower index of the second output array.

**u2:** (input) An integer value for the upper index of the second output array.

**res:** (output) The array of array with specified dimensions generated from the input data after verifying consistency.

*EXPRESS* specification:

```
*)
END_SCHEMA; -- end GEOMETRY schema
(*
```

# 5 Topology

The following EXPRESS declaration begins the **topology_schema** and identifies the necessary external references.

*EXPRESS* specification:

```
*)
SCHEMA topology_schema;
  REFERENCE FROM geometry_schema;
  REFERENCE FROM representation_schema(representation_item);
(*
```

NOTES

1 – The schemas referenced above can be found in the following Parts of ISO 10303:

| | |
|---|---|
| geometry_schema | Clause 4 of this part of ISO 10303 |
| representation_schema | ISO 10303-43 |

2 – See annex D, figures D.13-D.15, for a graphical presentation of this schema.



## 5.1 Introduction

The topology resource model has its basis in boundary representation solid modelling but can be used in any other application where an explicit method is required to represent connectivity.

## 5.2 Fundamental concepts and assumptions

The topological entities, **vertex**, **edge** etc., specified here have been defined independently of any use that may be made of them. Minimal constraints have been placed on each entity with the intention that any additional constraints will be specified by the using entity or by a defined context in which the entity is used. The intent is to avoid limiting the context or the use made of the entities.

The topological entities have been defined in a hierarchical manner with the **vertex** being the primitive entity. That is, all other topological entities are defined either directly or indirectly in terms of vertices.

Each entity has its own set of constraints. A higher-level entity may impose constraints on a lower-level entity. At the higher level, the constraints on the lower-level entity are the sum of the constraints imposed by each entity in the chain between the higher- and lower-level entities. The basic topological structures in order of increasing complexity are **vertex**, **edge**, **path**, **loop**, **face** and **shell**. In addition to the high-level structured topological entities **open_shell** and **closed_shell**, which are specialised subtypes of **connected_face_set**, the topology section includes the **connected_edge_set** and the general **connected_face_set**. These two entities are designed for the communication of collections of topological data where the constraints applied to shell are inappropriate.

The **poly_loop** is a loop with straight and coplanar edges and is defined as an ordered list of points. The **poly_loop** entity is used for the communication of faceted B-rep models.

Many functions ensure consistency of the topology models by applying topological and geometric constraints to entities.

### 5.2.1 Geometric associations

Many of the topological entities have a specialised subtype which enables them to be associated with geometric data. This association will be essential when communicating boundary representation solid models. The specialised subtypes of **vertex**, **edge** and **face** are **vertex_point**, **edge_curve**, and **face_surface** respectively. For the **edge_curve** and **face_surface** the relationship between the geometric sense and the topological sense of the associated entities is also recorded. The key concept relating geometry to topology is the domain. The domain of a **point**, **curve**, or **surface** is just that point, curve, or surface. The domain of a **vertex**, **edge**, or **face** is the corresponding point, curve or surface. The domain of a **loop** or **path** is the union of the domains of all the vertices and edges in the **loop** or **path**. (Except in the case of a vertex loop, this is a curve.) The domain of a shell is the union of the domains of all the vertices, edges, and faces in the shell. (For a **closed_shell** or **open_shell**, this is a surface.) The domain of a solid model is the region of space it occupies. The domain of a set or list is the union of the domains of the elements of that set or list. Wherever in this standard a geometrical concept such as connectedness or finiteness is discussed in relation to an entity, it is understood that the concept applies to the domain of that entity.



A key concept in describing domains is the idea of a manifold. Intuitively, a domain is a $d$-manifold if it is locally indistinguishable from $d$-dimensional Euclidean space. This means that the dimensionality is the same at each mathematical point, and self- intersections are prohibited. As defined in this standard, curves and surfaces may contain self-intersections, and hence need not be manifolds. However, the part of a curve or surface that corresponds to the domain of a topological entity such as an edge or face shall be a manifold.

As used in this standard, the terms "manifold", "boundary" , and " manifold with boundary" are identical to the usual mathematical definitions. A manifold with boundary differs from a manifold in that the boundary is allowed, but not required, to be non-empty.

A 1-manifold is a non-self-intersecting curve which does not include either of its end points. Examples of 1-manifolds are the real line and the unit circle. A "Y"-shaped figure is not a 1-manifold, and neither is the closed unit interval. A 2-manifold is a non-self-intersecting surface which does not include boundary curves. Examples of 2-manifolds include the unit sphere and the open disk $\{(x, y, 0) : x^2 + y^2 < 1\}$. The closed disk $\{(x, y, 0) : x^2 + y^2 \leq 1\}$ is not a manifold. The domains of edges and paths, if present, are 1-manifolds. The domains of faces and closed shells, if present, are 2-manifolds.

Any curve which does not self-intersect is a 1-manifold with boundary. The closed disk $\{(x, y, 0) : x^2 + y^2 \leq 1\}$ is a 2-manifold with boundary. The domain of an open shell, if present, is a 2-manifold with boundary. The domain of a manifold solid boundary representation or a faceted manifold boundary representation is a 3-manifold with boundary.

The boundary of a $d$-manifold with boundary is a $(d-1)$-manifold. For example, the boundary of a curve is the set of 0, 1, or 2 end points contained in that curve. The boundary of the closed disk $\{(x, y, 0) : x^2 + y^2 \leq 1\}$ is the unit circle. The boundary of the domain of an open shell is the domain of the set of loops that bound holes in the shell. The boundary of a manifold solid boundary representation or a faceted manifold boundary representation is the domain of the set of bounding shells.

Curves and surfaces which are manifolds with boundary are classified as either open or closed. The terms "open" and "closed", when applied to curves or surfaces in this standard, should not be confused with the notions of "open set" or "closed set" from point set topology. The term "closed surface" is identical to the usual definition of a closed, connected, orientable 2-manifold. Examples of a closed surface are a sphere and a torus. The domain of a closed shell, if present, is a closed surface. Examples of open surfaces are an infinite plane, or a surface with one or more holes. The domain of an open shell, if present, is an open surface.

All closed surfaces that are physically manufacturable are orientable. Face domains, because they are always embeddable in the plane, are orientable. Open surfaces need not be orientable. For example, the Möbius strip is an open surface. Also, some manifolds are neither open nor closed as defined in this standard. The Klein bottle is an example. It is finite and its boundary is empty, but the surface is not orientable, and hence does not divide space into two regions. However, the domain of an open shell as defined in this standard must be orientable.

The term "genus" refers to an integer-valued function used to classify topological properties of an entity. This standard defines two different types of genus.

For an entity which can be described as a graph of edges and vertices, for example a loop, path, or wire shell, genus is equivalent to the standard technical term "cycle rank" in graph theory. It is *not* equivalent to the standard usage of the term "genus" in graph theory. Intuitively, it measures the number of independent cycles in a graph. For example, a graph with exactly one vertex, joined to itself by $n$ self-loops, has genus $n$.

The genus of a closed surface $X$ is the number of handles that must be added to a sphere to produce a surface homeomorphic to $X$. For example, the genus of a sphere is 0, and the genus of a torus is 1. This is identical to the standard technical term "genus of a surface" from algebraic topology. Adding a handle to a closed surface is the operation that corresponds to drilling a tunnel through the three-dimensional volume bounded by that surface. This can be viewed as cutting out two disks and connecting their boundaries with a cylindrical tube. Handles should not be confused with holes. As used in this standard, the term "hole" corresponds to the intuitive notion of punching a hole in a two-dimensional surface.

The surface genus definition is extended to orientable open surfaces as follows. Fill in every hole in the domain with a disk. The resulting surface is a closed surface, for which genus is already defined. Use this number for the genus of the open surface.



### 5.2.2 Associations with parameter space geometry

A fundamental assumption in this clause is that the topology being defined is that of model space. The geometry of curves and points can also be defined in parameter space but, in general, the topological structure of, for example a **face**, will not be the same in the parametric space of the underlying surface as it is in model space.

Parametric space modelling systems differ from real space systems in the methodology used to associate geometry to topology. Parametric space modelling systems typically associate a different parametric space curve with each edge use (i.e., **oriented_edge**). Every one of the parametric space curves associated with a given edge (by way of an edge use) describe the same point set in real space. The parametric space curves are defined in different parametric spaces. The parametric spaces are the surfaces which underlay the faces bordering on the edge. In a manifold solid the geometry of every **edge** is define twice, once for each of the two **face**s which border on that **edge**.

Associating a parametric space curve with each edge use extends naturally to the use of degenerate edges (i.e., edges with zero length in real space). For example, a parametric space modelling system wants to represent a face that is triangular in real space as a square in parametric space. A straight forward way to do this is to represent one of the triangular face's vertices as a degenerate edge (but having two vertices); then there is a one-to-one mapping between edges in real space and model space. The degenerate edge has zero length in real space, but greater than zero length in parametric space. Degenerate edges also may be used for creating bounds around singularities such as the apex of a cone.

Real space modelling systems do not associate parametric space curves with each edge use nor do they allow degenerate edges. Since the parametric space modelling systems treatment of topology is an implementation convenience, this standard requires the use of real space topology. The parametric space modelling system's unique information requirements are satisfied using techniques at the geometric level.



#### 5.2.2.1 Edge_curve associations with parametric space curves.

Techniques that can be used to associate parametric space curves with an **edge_curve** are:

a) The **edge_geometry** attribute of an **edge_curve** may reference directly one **pcurve**, then only one **pcurve** is associated with that **edge_curve**.

b) The **edge_geometry** attribute of an **edge_curve** can reference a **surface_curve**, or a subtype of **surface_curve**; then associated with that **edge_curve** are the **pcurves** (one or two) referenced by the **associated_geometry** attribute of the **surface_curve**. The curve referenced by the **curve_3d** attribute of the **surface_curve** is also associated with the **edge_curve** but that curve cannot be a parametric space curve and represents the model space geometry of the **edge**.

c) The **edge_geometry** attribute of an **edge_curve** can reference a curve (not a **pcurve**), then associated with the **edge_curve** are the **pcurves** (zero or more) referenced by the **associated_geometry** attribute of every **surface_curve** whose **curve_3d** attribute references the same curve (i.e., is instance equal to, :=:) as the **edge_geometry** attribute of the **edge_curve**.

These techniques are formally defined in EXPRESS as the function **edge_curve_pcurves** which can be used to determine all the parametric space curves associated with a particular **edge**.

NOTES

1 – For applications where the real space modelling systems are not required to understand parametric space curves, the parametric space modelling systems should be required to use only the third technique described above. Then, even if the **pcurves** are ignored, the real space modelling system will have the correct geometry associated with all **edge_curves**.

2 – Given the **pcurves** of an **edge_curve**, determining which **oriented_edge** a pcurve shall be associated with is a matter of matching (:=:) the **basis_surface** of the **pcurve** with the **face_geometry** of the face bound by that **oriented_edge**. If two or more **pcurves** are associated with the same **edge_curve** and are defined in the parametric space of the same surface, determining which **oriented_edge** the **pcurve** is associated with requires checking connectivity of the **pcurves** in parametric space.

### 5.2.3 Graphs, cycles, and traversals

A connected component of a graph is a connected subset of the graph which is not contained in any larger connected subset. We denote by $M$ the *multiplicity* of a graph, that is, the number of connected components. Thus, a graph is connected if and only if $M = 1$.

Each component of a graph can be completely traversed, starting and ending at the same vertex, such that every edge is traversed exactly twice, once in each direction, and every vertex is "passed through" the same number of times as there are edges using the vertex. If an (edge + edge traversal direction) is considered as a unit, each unique (edge + direction) combination shall occur once and only once in the traversal of a graph. During the traversal of a graph it will be found that there are one or more sets of alternating vertices and (edge + direction) units that form closed cycles.

The symbol $G$ will denote the *graph genus*, which is, intuitively, the number of independent cycles in the graph. (Technically, $G$ is the rank of the fundamental group of the graph.)

Every graph satisfies the following Euler equation

$$(\mathcal{V} - \mathcal{E}) - (M - G) = 0 \quad (1)$$

where $\mathcal{V}$ and $\mathcal{E}$ are the numbers of unique vertices and edges in the graph.

NOTE – The following *graph traversal* algorithm, [E-4], may be used to traverse a graph and compute $M$ and $G$.

a) Set $M$ and $G$ to zero.

b) Start at any (unvisited) vertex. If there is no unvisited vertex, STOP. Mark the vertex as *visited*. Increment $M$. Traverse any edge at the vertex, marking the edge with the travel direction.

c) After traversing an edge $PQ$ to reach the vertex $Q$, do the following:

– When reaching a vertex for the first time, mark the edge just travelled as the *advent edge* of the vertex. The advent edge is marked so that it can only be selected once in this direction.

– Mark the vertex as *visited*.

– If this is the first traversal of the edge and the vertex $Q$ has previously been visited, increment $G$.

– Select an exit edge from the vertex according to the following rules:

(1) No edge may be selected that has previously been traversed in the direction away from the vertex $Q$.

(2) Select any edge, except the advent edge of $Q$, that meets rule (c1).

(3) If no edge meets rule (c2), select the advent edge.

– Traverse the selected exit edge and mark it with the travel direction.

d) If no edge was selected in the previous step, go to step b, else go to step c.

## 5.3 topology_schema type definitions

### 5.3.1 shell

This type collects together, for reference when constructing more complex models, the subtypes which have the characteristics of a shell. A **shell** is a connected object of fixed dimensionality $d = 0, 1$, or 2, typically used to bound a region. The domain of a shell, if present, includes its bounds and $0 \leq \Xi < \infty$. A shell of dimensionality 0 is represented by a graph consisting of a single vertex. The vertex shall not have any associated edges.

A shell of dimensionality 1 is represented by a connected graph of dimensionality 1.

A shell of dimensionality 2 is a topological entity constructed by joining faces along edges. Its domain, if present, is a connected, orientable 2-manifold with boundary, that is, a connected, oriented, finite, non-self-intersecting surface, which may be closed or open.

*EXPRESS* specification:

```
*)
TYPE shell = SELECT
  (vertex_shell,
   wire_shell,
   open_shell,
   closed_shell);
END_TYPE;
(*
```

### 5.3.2 reversible_topology_item

This select type specifies all the topological representation items which can participate in the operation of reversing their orientation. This type is used by the function **conditional_reverse**.

*EXPRESS* specification:

```
*)
TYPE reversible_topology_item = SELECT
  (edge,
   path,
   face,
   face_bound,
   closed_shell,
   open_shell);
END_TYPE;
(*
```

### 5.3.3 list_of_reversible_topology_item

This special type defines a list of reversible topology items; it is used by the function **list_of_topology_reversed**.

*EXPRESS* specification:

```
*)
TYPE list_of_reversible_topology_item =
                              LIST [0:?] of reversible_topology_item;
END_TYPE;
(*
```

### 5.3.4 set_of_reversible_topology_item

This special type defines a set of reversible topology items; it is used by the function **set_of_topology_reversed**.

*EXPRESS* specification:

```
*)
TYPE set_of_reversible_topology_item =
                    SET [0:?] of reversible_topology_item;
END_TYPE;
(*
```

### 5.3.5 reversible_topology

This select type identifies all types of reversible topology items; it is used by the function **topology_reversed**.

*EXPRESS* specification:

```
*)
TYPE reversible_topology = SELECT
          (reversible_topology_item,
           list_of_reversible_topology_item,
           set_of_reversible_topology_item);
END_TYPE;
(*
```

## 5.4 topology_schema entity definitions

This clause contains all the entity definitions used in the topology schema.

### 5.4.1 topological_representation_item

A **topological_representation_item** represents the topology, or connectivity, of entities which make up the representation of an object. The **topological_representation_item** is the supertype for all the representation items in the topology schema.

*EXPRESS* specification:

```
*)
ENTITY topological_representation_item
  SUPERTYPE OF (ONEOF(vertex, edge, face_bound, face, vertex_shell,
                wire_shell, connected_edge_set, connected_face_set,
                 (loop ANDOR path)))
  SUBTYPE OF (representation_item);
END_ENTITY;
(*
```

Informal propositions:

**IP1:** For each **topological_representation_item**, consider the set of **vertex_points**, **edge_curves**, and **face_surfaces** that are referenced, either directly or recursively, from that **topological_representation_item**. (Do not include in this set oriented edges or faces, but do include the non-oriented edges and faces on which they are based.) Then no two distinct elements in this set shall have domains that intersect.

### 5.4.2 vertex

A **vertex** is the topological construct corresponding to a point. It has dimensionality 0 and extent 0. The domain of a vertex, if present, is a point in m dimensional real space $R^m$; this is represented by the **vertex_point** subtype.

*EXPRESS* specification:

```
*)
ENTITY vertex
  SUBTYPE OF (topological_representation_item);
END_ENTITY;
(*
```

Informal propositions:

**IP1:** The **vertex** has dimensionality 0. This is a fundamental property of the vertex.

**IP2:** The extent of a **vertex** is defined to be zero.

### 5.4.3 vertex_point

A vertex point is a vertex which has its geometry defined as a point.

*EXPRESS* specification:

```
*)
ENTITY vertex_point
SUBTYPE OF(vertex,geometric_representation_item);
  vertex_geometry : point;
END_ENTITY;
(*
```

Attribute definitions:

**vertex_geometry:** The geometric point which defines the position in geometric space of the vertex.

Informal propositions:

**IP1:** The domain of the vertex is formally defined to be the domain of its **vertex_geometry**.

### 5.4.4 edge

An **edge** is the topological construct corresponding to the connection between two vertices. More abstractly, it may stand for a logical relationship between the two vertices. The domain of an edge, if present, is a finite, non-self-intersecting open curve in $R^m$, that is, a connected 1-dimensional manifold. The bounds of an **edge** are two vertices, which need not be distinct. The edge is oriented by choosing its traversal direction to run from the first to the second vertex. If the two vertices are the same, the edge is a self-loop. The domain of the edge does not include its bounds, and $0 < \Xi < \infty$. Associated with an edge may be a geometric **curve** to locate the edge in a coordinate space; this is represented by the **edge curve** subtype. The curve shall be finite and non-self-intersecting within the domain of the edge. An **edge** is a graph, so its multiplicity $M$ and graph genus $G^e$ may be determined by the graph traversal algorithm. Since $M = \mathcal{E} = 1$, the Euler equation (1) reduces in this case to

$$\mathcal{V} - (2 - G^e) = 0 \qquad (2)$$

where $\mathcal{V} = 1$ or 2, and $G^e = 1$ or 0.

Specifically, the topological edge defining data shall satisfy:

– An edge has two vertices,

$$|E[V]| = 2$$

– The vertices need not be distinct,

$$1 \leq |E\{V\}| \leq 2$$

– Equation 2 shall hold

$$|E\{V\}| - 2 + G^e = 0$$

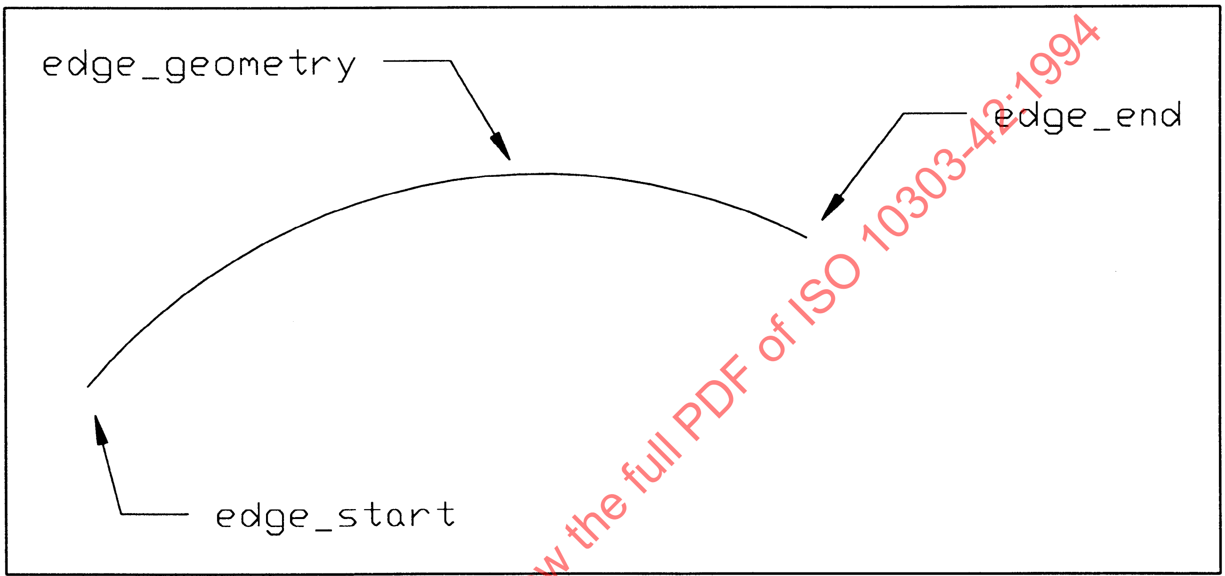


**Figure 11 – Edge curve**



*EXPRESS* specification:

```
*)
ENTITY edge
  SUPERTYPE OF(ONEOF(edge_curve, oriented_edge))
  SUBTYPE OF (topological_representation_item);
  edge_start : vertex;
  edge_end   : vertex;
END_ENTITY;
(*
```

Attribute definitions:

**edge_start:** Start point (**vertex**) of the **edge**.

**edge_end:** End point (**vertex**) of the **edge**. The same **vertex** can be used for both **edge_start** and **edge_end**.

Informal propositions:

**IP1:** The **edge** has dimensionality 1.

**IP2:** The extent of an **edge** shall be finite and nonzero.

### 5.4.5 edge_curve

An **edge_curve** is a special subtype of edge which has its geometry fully defined. The geometry is defined by associating the edge with a curve which may be unbounded. As the topological and geometric directions may be opposed, an indicator (**same_sense**) is used to identify whether the edge and curve directions agree or are opposed. The logical value indicates whether the **curve** direction agrees with (TRUE) or is in the opposite direction (FALSE) to the **edge** direction. Any geometry associated with the vertices of the edge shall be consistent with the edge geometry. Multiple edges can reference the same curve.

*EXPRESS* specification:

```
*)
ENTITY edge_curve
  SUBTYPE OF(edge,geometric_representation_item);
  edge_geometry : curve;
  same_sense    : BOOLEAN;
END_ENTITY;
(*
```

Attribute definitions:

**edge_geometry:** The curve which defines the shape and spatial location of the edge. This curve may be unbounded and is implicitly trimmed by the vertices of the edge; this defines the edge domain.

**same_sense:** This logical flag indicates whether (TRUE), or not (FALSE) the senses of the **edge** and the **curve** defining the edge geometry are the same. The sense of an edge is from the edge start vertex to the edge end vertex; the sense of a curve is in the direction of increasing parameter.

NOTE – See figure 11 for illustration of attributes.

Informal propositions:

**IP1:** The domain of the **edge_curve** is formally defined to be the domain of its **edge_geometry** as trimmed by the vertices. This domain does not include the vertices.

**IP2:** An **edge_curve** has non-zero finite extent.

**IP3:** An **edge_curve** is a manifold.

**IP4:** An **edge_curve** is arcwise connected.

**IP5:** The edge start is not part of the edge domain.

**IP6:** The edge end is not part of the edge domain.

**IP7:** Vertex geometry shall be consistent with edge geometry.

### 5.4.6 oriented_edge

An **oriented_edge** is an **edge** constructed from another **edge** and contains a Boolean orientation flag to indicate whether or not the orientation of the constructed **edge** agrees with the orientation of the original **edge**. Except for possible re-orientation, the **oriented_edge** is equivalent to the original **edge**.

NOTE – A common practice in solid modelling systems is to have an entity that represents the "use" or "traversal" of an **edge**. This "use" entity explicitly represents the requirement in a manifold solid that each edge must be traversed exactly twice, once in each direction. The "use" functionality is provided by the **edge** subtype **oriented_edge**.

*EXPRESS* specification:

```
*)
ENTITY oriented_edge
  SUBTYPE OF (edge);
  edge_element : edge;
  orientation  : BOOLEAN;
DERIVE
  SELF\edge.edge_start : vertex := boolean_choose (SELF.orientation,
                                          SELF.edge_element.edge_start,
                                          SELF.edge_element.edge_end);
  SELF\edge.edge_end   : vertex := boolean_choose (SELF.orientation,
                                          SELF.edge_element.edge_end,
                                          SELF.edge_element.edge_start);
WHERE
  WR1: NOT ('TOPOLOGY_SCHEMA.ORIENTED_EDGE' IN TYPEOF (SELF.edge_element));
END_ENTITY;
(*
```

Attribute definitions:

**edge_element: edge** entity used to construct this **oriented_edge**.

**orientation:** BOOLEAN. If TRUE, the topological orientation as used coincides with the orientation, from start vertex to end vertex, of the **edge_element**.

**edge_start:** The start vertex of the oriented edge. This is derived from the vertices of the **edge_element** after taking account of the **orientation**

**edge_end:** The end vertex of the oriented edge. This is derived from the vertices of the **edge_element** after taking account of the **orientation**

Formal propositions:

**WR1:** The **edge_element** shall not be an **oriented_edge**.

### 5.4.7 path

A **path** is a topological entity consisting of an ordered collection of **oriented_edges**, such that the **edge_start** vertex of each edge coincides with the **edge_end** of its predecessor. The path is ordered from the **edge_start** of its first **oriented_edge** to the **edge_end** of its last **oriented_edge**. The Boolean value **orientation** in the oriented edge indicates whether the edge direction agrees with the direction of the path (TRUE) or is in the opposite direction (FALSE).

An individual **edge** can only be referenced once by an individual **path**.

An **edge** can be referenced by multiple **paths**. An **edge** can exist independently of a **path**.

*EXPRESS* specification:

```
*)
ENTITY path
  SUPERTYPE OF (ONEOF(open_path, edge_loop, oriented_path))
  SUBTYPE OF (topological_representation_item);
  edge_list  : LIST [1:?] OF UNIQUE oriented_edge;
WHERE
  WR1: path_head_to_tail(SELF);
END_ENTITY;
(*
```

Attribute definitions:

**edge_list:** List of **oriented_edge** entities which are concatenated together to form this **path**.

Formal propositions:

**WR1:** The end vertex of each **oriented_edge** shall be the same as the start vertex of its successor.

Informal propositions:

**IP1:** A **path** has dimensionality 1.

**IP2:** A **path** is arcwise connected.

**IP3:** The edges of the path do not intersect except at common vertices.

**IP4:** A path has a finite, non-zero extent.

**IP5:** No **path** shall include two oriented edges with the same edge element and the same orientation.

### 5.4.8 oriented_path

An **oriented_path** is a **path** constructed from another **path** and contains a Boolean orientation flag to indicate whether or not the orientation of the constructed **path** agrees with the orientation of the original **path**. Except for perhaps orientation, the **oriented_path** is equivalent to the other **path**.

*EXPRESS* specification:

```
*)
ENTITY oriented_path
  SUBTYPE OF (path);
```

```
  path_element : path;
  orientation  : BOOLEAN;
DERIVE
  SELF\path.edge_list : LIST [1:?] OF UNIQUE oriented_edge
                        := conditional_reverse(SELF.orientation,
                                  SELF.path_element.edge_list);
WHERE
  WR1: NOT ('TOPOLOGY_SCHEMA.ORIENTED_PATH' IN TYPEOF (SELF.path_element));
END_ENTITY;
(*
```

Attribute definitions:

**path_element:** **path** entity used to construct this **oriented_path**.

**orientation:** BOOLEAN. If TRUE, the topological orientation as used coincides with the orientation of the **path_element**.

**edge_list:** The list of **oriented_edges** which form the **oriented_path**. This list is derived from the **path_element** after taking account of the **orientation** attribute.

Formal propositions:

**WR1:** The **path_element** shall not be an **oriented_path**.

### 5.4.9 open_path

An **open_path** is a special subtype of **path** such that a traversal of the path visits each of its vertices exactly once. In particular, the start vertex and end vertex are different. An **open_path** is a graph for which $M = 1$ and $G^p = 0$, so the Euler equation (1) reduces in this case to

$$(\mathcal{V} - \mathcal{E}) - 1 = 0 \tag{3}$$

where $\mathcal{V}$ and $\mathcal{E}$ are the number of unique vertices and edges in the path. Specifically, the topological attributes of a **path** shall meet the following constraints

– The edges in the Path are unique,

$$(P)[E] = (P)\{E\}$$

– In the list $((P)[E])[V]$, two vertices appear once only and every other vertex appears exactly twice.

– The graph genus of the path is zero.

– Equation (3) is interpreted as

$$|((P)[E])\{V\}| - |(P)\{E\}| - 1 = 0$$

*EXPRESS* specification:

```
*)
ENTITY open_path
  SUBTYPE OF (path);
DERIVE
  ne : INTEGER := SIZEOF(SELF\path.edge_list);
WHERE
  WR1: (SELF\path.edge_list[1].edge_element.edge_start) :<>:
                       (SELF\path.edge_list[ne].edge_element.edge_end);
END_ENTITY;
(*
```

Attribute definitions:

**ne:** The number of elements in the edge list of the path supertype.

Formal propositions:

**WR1:** The start vertex of the first edge shall not coincide with the end vertex of the last edge.

Informal propositions:

**IP1:** An **open_path** visits its **vertex**s exactly once. This implies that if a list of vertices is constructed from the edge data the first and last vertex will occur once in this list and all other vertices will occur twice.



## 5.4.10 loop

A **loop** is a topological entity constructed from a single vertex, or by stringing together connected (oriented) edges, or linear segments beginning and ending at the same vertex. A loop has dimensionality 0 or 1. The domain of a 0-dimensional loop is a single point. The domain of a 1-dimensional loop is a connected, oriented curve, but need not be a manifold. As the loop is a cycle, the location of its beginning/ending point is arbitrary. The domain of the loop includes its bounds, and $0 \leq \Xi < \infty$.

A loop is represented by a single vertex, or by an ordered collection of **oriented_edges**, or by an ordered collection of points.

A loop is a graph, so $M$ and the graph genus $G^l$ may be determined by the graph traversal algorithm. Since $M = 1$, the Euler equation (1) reduces in this case to

$$(\mathcal{V} - \mathcal{E}_l) - (1 - G^l) = 0 \quad (4)$$

where $\mathcal{V}$ and $\mathcal{E}_l$ are the number of unique vertices and oriented edges in the loop and $G^l$ is the genus of the loop.

*EXPRESS* specification:

```
*)
ENTITY loop
  SUPERTYPE OF (ONEOF(vertex_loop, edge_loop, poly_loop))
  SUBTYPE OF (topological_representation_item);
```

```
END_ENTITY;
(*
```

Informal propositions:

**IP1:** A **loop** has a finite, or, in the case of the **vertex_loop**, zero extent.

**IP2:** A **loop** describes a closed (topological) curve with coincident start and end vertices.

### 5.4.11 vertex_loop

A **vertex_loop** is a **loop** of zero genus consisting of a single **vertex**. A **vertex** can exist independently of a **vertex_loop**. The topological data shall satisfy the following constraint:

– Equation (4) (see 5.4.10) shall be satisfied

$$|(L)\{V\}| - 1 = 0$$

*EXPRESS* specification:

```
*)
ENTITY vertex_loop
  SUBTYPE OF (loop);
  loop_vertex : vertex;
END_ENTITY;
(*
```

Attribute definitions:

**loop_vertex:** The **vertex** which defines the entire **loop**.

Informal propositions:

**IP1:** A **vertex_loop** has zero extent and dimensionality.

**IP2:** The **vertex_loop** has genus 0.

### 5.4.12 edge_loop

An **edge_loop** is a **loop** with nonzero extent. It is a **path** in which the start and end vertices are the same. Its domain, if present, is a closed curve. An **edge_loop** may overlap itself.

*EXPRESS* specification:

```
*)
ENTITY edge_loop
  SUBTYPE OF (loop,path);
DERIVE
  ne : INTEGER := SIZEOF(SELF\path.edge_list);
WHERE
  WR1: (SELF\path.edge_list[1].edge_start) :=:
       (SELF\path.edge_list[ne].edge_end);
```

```
END_ENTITY;
(*
```

Attribute definitions:

**ne:** The number of elements in the edge list of the path supertype.

Formal propositions:

**WR1:** The start vertex of the first edge shall be the same as the end vertex of the last edge. This ensures that the path is closed to form a loop.

Informal propositions:

**IP1:** The genus of the **edge_loop** shall be 1 or greater.

**IP2:** The Euler formula (see equation (4) shall be satisfied:

$$\text{(number of vertices)} + \text{genus} - \text{(number of edges)} = 1;$$

**IP3:** No **edge** may be referenced more than once by the same **edge_loop** with the same **orientation**.



### 5.4.13 poly_loop

A **poly_loop** is a loop with straight edges bounding a planar region in space. A **poly_loop** is a **loop** of genus 1 where the loop is represented by an ordered coplanar collection of **points** forming the vertices of the loop. The loop is composed of straight line segments joining a point in the collection to the succeeding point in the collection. The closing segment is from the last to the first point in the collection. The direction of the loop is in the direction of the line segments. Unlike the **edge_loop** entity, the edges of the **poly_loop** are implicitly defined by the **polygon** points.

NOTE – This entity exists primarily to facilitate the efficient communication of faceted B-rep models.

A **poly_loop** shall conform to the following topological constraints:

- The loop has a genus of one.
- Equation (4) (see 5.4.10) shall be satisfied

$$|(L)\{V\}| - |(L)\{E_l\}| = 0$$

*EXPRESS* specification:

```
*)
ENTITY poly_loop
  SUBTYPE OF (loop,geometric_representation_item);
  polygon : LIST [3:?] OF UNIQUE cartesian_point;
END_ENTITY;
(*
```

Attribute definitions:

**polygon:** List of **points** defining the loop. There are no repeated **points** in the list.

Informal propositions:

**IP1:** All the points in the **polygon** defining the **poly_loop** shall be coplanar.

**IP2:** The implicit edges of the **poly_loop** shall not intersect each other. The implicit edges are the straight lines joining consecutive **points** in the **polygon**.

NOTE – The polyloop has vertices and **oriented_edges** which are implicitly created. If, for example, A and B are consecutive points in the **polygon** list, there is an implicit **oriented_edge** from vertex point A to vertex point B with orientation value TRUE. It is assumed that when the higher level entities such as shell and B-rep require checks on edge usage that this check will recognise, for example, a straight oriented edge from point B to point A with orientation TRUE as equal to an oriented edge from A to B with orientation FALSE.



## 5.4.14 face_bound

A **face_bound** is a loop which is intended to be used for bounding a face.

*EXPRESS* specification:

```
*)
ENTITY face_bound
  SUBTYPE OF(topological_representation_item);
  bound       :  loop;
  orientation :  BOOLEAN;
END_ENTITY;
(*
```

Attribute definitions:

**bound:** The loop which will be used as a face boundary.

**orientation:** This indicates whether (TRUE), or not (FALSE) the loop has the same sense when used to bound the face as when first defined. If **orientation** is FALSE, the senses of all its component oriented edges are implicitly reversed when used in the face.

## 5.4.15 face_outer_bound

A **face_outer_bound** is a special subtype of **face_bound** which carries the additional semantics of defining an outer boundary on the face. No more than one boundary of a **face** shall be of this type.

*EXPRESS* specification:

```
*)
ENTITY face_outer_bound
SUBTYPE OF (face_bound);
END_ENTITY;
(*
```

## 5.4.16 face

A **face** is a topological entity of dimensionality 2 corresponding to the intuitive notion of a piece of surface bounded by loops. Its domain, if present, is an oriented, connected, finite 2-manifold in $R^m$. A face domain shall not have handles, but it may have holes, each hole bounded by a loop. The domain of the underlying geometry of the face, if present, does not contain its bounds, and $0 < \Xi < \infty$. A face is represented by its bounding loops, which are defined as **face_bounds**. A face shall have at least one bound, and the bounds shall be distinct and shall not intersect. One **loop** is optionally distinguished, using the **face_outer_bound** subtype, as the "outer" loop of the face. If so, it establishes a preferred way of embedding the face domain in the plane, in which the other bounding loops of the face are "inside" the outer loop. Because the face domain is arcwise connected, no inner loop shall contain any other loop. This is true regardless of which embedding in the plane is chosen.

A geometric surface may be associated with the face. This may be done explicitly through the **face_surface** subtype, or implicitly if the faces are defined by **poly_loops**. In the latter case, the surface is the plane containing the points of the **poly_loops**. In either case, a topological normal **n** is associated with the face, such that the cross product$n \times t$ points toward the interior of the face, where **t** is the tangent to a bounding loop. That is, each loop runs counter-clockwise around the face when viewed from above, if we consider the normal **n** to point up. Each loop is associated through a **face_bound** entity with a Boolean flag to signify whether the loop direction is oriented correctly with respect to the face normal (TRUE) or should be reversed (FALSE).

For a face of the subtype **face_surface**, the topological normal n is defined from the normal of the underlying surface, together with the Boolean attribute **same_sense**, and this in turn, determines on which side of the loop the face interior lies, using the cross-product rule described above. The situation is different for a face on an implicit planar surface, such as one defined by **poly_loops**, which has no unique surface normal. Since the face and its bounding loops lie in a plane, the outer loop can always be found without ambiguity. Since the face is required to be finite, the face interior must lie inside the outer loop, and outside each of the remaining loops. These conditions, together with the specified loop orientations, define the topological normal n using the cross-product rule described above. All **poly_loop** orientations for a given face shall produce the same value for **n**.

The edges and vertices referenced by the loops of a face form a graph, of which the individual loops are the connected components. The Euler equation (1) for this graph becomes:

$$(\mathcal{V} - \mathcal{E}) - (\mathcal{L} - \sum_{i=1}^{L}(G_i^l)) = 0 \quad (5)$$

where $G_i^l$ is the graph genus of the $i$'th loop.

More specifically, the following topological constraints shall be met:

- The loops are unique

$$(F)\{L\} = (F)[L]$$

- In the list $((F)[L])[E]$ an individual edge occurs no more than twice.

– Each **oriented_edge** shall be unique

$$((F)[L])\{E_l\} = ((F)[L])[E_l]$$

– Equation (5) shall be satisfied

$$|(((F)[L^e])\{E\})\{V\}| + |((F)[L^v])\{V\}| - |((F)[L])\{E\}| - |(F)[L]| + \sum G^l = 0$$

*EXPRESS* specification:

```
*)
ENTITY face
  SUPERTYPE OF(ONEOF(face_surface, subface, oriented_face))
  SUBTYPE OF (topological_representation_item);
  bounds : SET[1:?] OF face_bound;
WHERE
  WR1: NOT (mixed_loop_type_set(list_to_set(list_face_loops(SELF))));
  WR2: SIZEOF(QUERY(temp <* bounds | 'TOPOLOGY_SCHEMA.FACE_OUTER_BOUND' IN
                                             TYPEOF(temp))) <= 1;
END_ENTITY;
(*
```

Attribute definitions:

**bounds:** Boundaries of the **face**; no more than one of these shall be a **face_outer_bound**.

NOTE – For some types of closed or partially closed surfaces, it may not be possible to identify a unique outer bound.

Formal propositions:

**WR1:** If any loop of the face is a poly loop, all loops of the face shall be poly loops.

**WR2:** At most, one of the **bounds** shall be of type **face_outer_bound.**

Informal propositions:

**IP1:** No edge shall be referenced by the face more than twice, or more than once in the same direction.

**IP2:** Distinct **face_bounds** of the **face** shall have no common vertices.

**IP3:** If geometry is present, distinct loops of the same face shall not intersect.

**IP4:** The face shall satisfy the Euler equation (see equation (5)
): (number of vertices) – (number of edges) – (number of loops) + (sum of genus for loops) = 0.

**IP5:** Each **loop** referred to in **bounds** shall be unique.

### 5.4.17 face_surface

A **face_surface** is a subtype of face in which the geometry is defined by an associated surface. The portion of the surface used by the face shall be embeddable in the plane as an open disk, possibly with holes. However, the union of the face with the edges and vertices of its bounding loops need not be embeddable in the plane. It may, for example, cover an entire sphere or torus. As both a face and a geometric surface have defined normal directions, a Boolean flag (the orientation attribute) is used to indicate whether the surface normal agrees with (TRUE) or is opposed to (FALSE) the face normal direction. The geometry associated with any component of the loops of the face shall be consistent with the surface geometry, in the sense that the domains of all the vertex points and edge curves are contained in the face geometry surface. A **surface** may be referenced by more than one **face_surface**.

*EXPRESS* specification:

```
*)
ENTITY face_surface
  SUBTYPE OF(face,geometric_representation_item);
  face_geometry :  surface;
  same_sense    :  BOOLEAN;
END_ENTITY;
(*
```

Attribute definitions:

**face_geometry:** The surface which defines the internal shape of the face. This surface may be unbounded. The domain of the face is defined by this surface and the bounding loops in the inherited attribute **SELF\face.bounds**.

**same_sense:** This flag indicates whether the sense of the surface normal agrees with (TRUE), or opposes (FALSE), the sense of the topological normal to the **face**.

Informal propositions:

**IP1:** The domain of the **face_surface** is formally defined to be the domain of its **face_geometry** as trimmed by the loops, this domain does not include the bounding loops.

**IP2:** A **face_surface** has nonzero finite extent.

**IP3:** A **face_surface** is a manifold.

**IP4:** A **face_surface** is arcwise connected.

**IP5:** A **face_surface** has surface genus 0.

**IP6:** The loops are not part of the face domain.

**IP7:** Loop geometry shall be consistent with face geometry. This implies that any **edge_curves** or **vertex_points** used in defining the loops bounding the **face_surface** shall lie on the **face_geometry**.

**IP8:** The loops of the face shall not intersect.

### 5.4.18 oriented_face

An **oriented_face** is a subtype of face which contains an additional orientation Boolean flag to indicate whether, or not, the sense of the oriented face agrees with its sense as originally defined in the face element.

EXPRESS specification:

```
*)
ENTITY oriented_face
  SUBTYPE OF (face);
  face_element : face;
  orientation  : BOOLEAN;
DERIVE
  SELF\face.bounds : SET[1:?] OF face_bound
         := conditional_reverse(SELF.orientation,SELF.face_element.bounds);
WHERE
  WR1: NOT ('TOPOLOGY_SCHEMA.ORIENTED_FACE' IN TYPEOF (SELF.face_element));
END_ENTITY;
(*
```

Attribute definitions:

**face_element:** **Face** entity used to construct this **oriented_face**.

**orientation:** The relationship of the topological orientation of this entity to that of the **face_element**. If TRUE, the topological orientation as used coincides with the orientation of the **face_element**.

**bounds:** The bounds of the **oriented_face** are derived from those of the **face_element** after taking account of the orientation which may reverse the direction of these bounds.

Formal propositions:

**WR1:** The **face_element** shall not be an **oriented_face**.

### 5.4.19 subface

A **subface** is a portion of the domain of a **face**, or another **subface**.

The topological constraints on a **subface** are the same as on a **face**.

EXPRESS specification:

```
*)
ENTITY subface
  SUBTYPE OF (face);
  parent_face   :  face;
WHERE
  WR1: NOT (mixed_loop_type_set(list_to_set(list_face_loops(SELF)) +
             list_to_set(list_face_loops(parent_face))));
END_ENTITY;
(*
```

Attribute definitions:

**parent_face:** The **face**, (or **subface**) which contains the **subface** being defined by **SELF\face.bounds**.

Formal propositions:

**WR1:** The type of **loops** in the **subface** shall match the type of **loops** in the **parent_face** entity.

Informal propositions:

**IP1:** The domain of the subface is formally defined to be the domain of the parent face, as trimmed by the loops of the subface.

**IP2:** All loops of the subface shall be contained in the union of the domain of the parent face and the domains of the parent face's bounding loops.

## 5.4.20 connected_face_set

A **connected_face_set** is a set of **faces** such that the domain of the faces together with their bounding edges and vertices is connected.

*EXPRESS* specification:

```
*)
ENTITY connected_face_set
  SUPERTYPE OF (ONEOF (closed_shell, open_shell))
  SUBTYPE OF (topological_representation_item);
  cfs_faces : SET [1:?] OF face;
END_ENTITY;
(*
```

Attribute definitions:

**cfs_faces:** Set of **faces** arcwise connected along common **edges** or **vertexs**.

Informal propositions:

**IP1:** The union of the domains of the **faces** and their bounding **loops** shall be arcwise connected.

## 5.4.21 vertex_shell

A **vertex_shell** is a **shell** consisting of a single **vertex_loop**. A **vertex_shell_extent** shall be unique.

A **vertex_loop** can only be used by a single **vertex_shell**.

A **vertex_loop** can exist independently of a **vertex_shell**.

*EXPRESS* specification:

```
*)
ENTITY vertex_shell
  SUBTYPE OF (topological_representation_item);
  vertex_shell_extent : vertex_loop;
END_ENTITY;
```

(*

Attribute definitions:

**vertex_shell_extent:** Single **vertex_loop** which constitutes the extent of this type of **shell.**

Informal propositions:

**IP1:** The extent and dimensionality of a **vertex_shell** are both zero.

**IP2:** The genus of a **vertex_shell** is 0.

## 5.4.22 wire_shell

A **wire_shell** is a **shell** of dimensionality 1. A wire shell can be regarded as a graph constructed of vertices and edges. However, it is not represented directly as a graph, but indirectly, as a set of loops. It is the union of the vertices and edges of these loops that form the graph. The domain of a wire shell, if present, is typically not a manifold.

Two restrictions are placed on the structure of a wire shell.

a) The graph as a whole shall be connected.

b) Each edge in the graph shall be referenced exactly twice by the set of loops.

NOTES

1 – Two main applications of wire shells are contemplated.

2 – Any connected graph can be written as a single loop obeying condition (b) by using the graph traversal algorithm. Such a graph may serve as a bound for a region.

3 – The set of loops referenced by the faces of a closed shell automatically obey condition (b), but need not be connected. However, the faces of a closed shell can always be subdivided in such a way that their loops form a connected graph, and hence a wire shell. Thus, wire shells can represent the "one-dimensional skeleta" of closed shells.

Writing $G^w$ for the graph genus, and setting the number of connected components $M = 1$, the Euler graph equation (1) becomes:

$$(\mathcal{V} - \mathcal{E}) - (1 - G^w) = 0 \qquad (6)$$



More specifically, the following topological constraints shall be met:

– The loops shall be unique

$$(S^w)\{L\} = (S^w)[L]$$

– Each edge shall either be referenced by two loops, or twice by a single loop. That is, in the list $((S^w)[L])[E]$, each edge appears exactly twice.

$$|((S^w)[L])[E]| = 2|((S^w)[L])\{E\}|$$

– Each oriented edge shall be unique.

$$((S^w)[L])\{E_l\} = ((S^w)[L])[E_l]$$

– Equation (6) shall be satisfied

$$|(((S^w)[L])\{E\})\{V\}| - |((S^w)[L])\{E\}| - 1 + G^w = 0$$

*EXPRESS* specification:

```
*)
ENTITY wire_shell
  SUBTYPE OF (topological_representation_item);
  wire_shell_extent : SET [1:?] OF loop;
WHERE
  WR1: NOT mixed_loop_type_set(wire_shell_extent);
END_ENTITY;
(*
```

Attribute definitions:

**wire_shell_extent:** List of **loops** defining the **shell**.

Formal propositions:

**WR1:** The loops making up the wire shell shall not be a mixture of **poly_loops** and other loop types.

Informal propositions:

**IP1:** The **wire_shell** has dimensionality 1.

**IP2:** The extent of the **wire_shell** is finite and greater than 0.

**IP3:** Each edge appears precisely twice in the wire shell with opposite orientations.

**IP4:** The Euler equation shall be satisfied.

**IP5:** The **loops** defining the **wire_shell_extent** do not intersect except at common **edges** or **vertexs**.

## 5.4.23 open_shell

An **open_shell** is a **shell** of dimensionality 2. Its domain, if present, is a finite, connected, oriented, 2-manifold with boundary, but is not a closed surface. It can be thought of as a **closed_shell** with one or more holes punched in it. The domain of an open shell satisfies $0 < \Xi < \infty$. An open shell is functionally more general than a **face** because its domain can have handles.

The shell is defined by a collection of **faces**, which may be **oriented_faces**. The sense of each face, after taking account of the orientation, shall agree with the shell normal as defined below.

The **orientation** can be supplied directly as a Boolean attribute of an **oriented_face**, or be defaulted to TRUE if the shell member is a **face** without the orientation attribute.

The following combinatorial restrictions on open shells and geometrical restrictions on their domains are designed, together with the informal propositions, to ensure that any domain associated with an open shell is an orientable manifold.

- Each face reference shall be unique.
- An **open_shell** shall have at least one **face**.
- A given **face** may exist in more than one **open_shell**.

The **boundary** of an open shell consists of the edges that are referenced only once by the **face_bounds** (loops) of its faces, together with all of their vertices. The domain of an open shell, if present, contains all edges and vertices of its faces.

NOTE – Note that this is slightly different from the definition of a face domain, which includes none of its bounds. For example, a face domain may exclude an isolated point or line segment. An open shell domain may not. (See the algorithm for computing $\mathcal{B}$ below.)

The surface genus and topological normal of an open shell are those that would be obtained by filling in the holes in its domain to produce a closed shell. The topological normal can also be derived from the face normals after taking account of their orientation. The following Euler equation is satisfied by open shells. It is the most general form of Euler equation for connected, orientable surfaces.

$$(\mathcal{V} - \mathcal{E} - \mathcal{L}_l + 2\mathcal{F}) - (2 - 2H - \mathcal{B}) = 0 \qquad (7)$$

where $\mathcal{V}, \mathcal{E}, \mathcal{L}_l, \mathcal{F}$ are, respectively, the numbers of distinct vertices, edges, face bounds, and faces, $H$ is the surface genus, and $\mathcal{B}$ is the number of holes. $\mathcal{B}$ can be determined directly from the graph of edges and vertices defining the bounds of the face, in the following manner:

- Delete all edges from the graph that are referenced twice by the face bounds of the face.
- Delete all vertices that have no associated edges.
- Compute $\mathcal{B}$ = the genus of the resulting graph.

If known a priori, the surface genus $H$ may be used to check equation (7) as an exact equality. Typically, this will not be the case, so equation (7) or some equivalent formulation shall be used to compute the genus. Since $H$ shall be a non-negative integer, this leads to the following inequality, a necessary condition for well-formed open shells.

$$\mathcal{V} - \mathcal{E} - \mathcal{L}_l + \mathcal{B} \text{ shall be even and } \leq 2 - 2\mathcal{F} \qquad (8)$$

Specifically, the following topological constraints shall be met:

- Each face in the shell is unique

$$(S^o)\{F\} = (S^o)[F]$$

- Each face bound in the shell is unique

$$((S^o)[F])\{L_l\} = ((S^o)[F])[L_l]$$

– Each **oriented_edge** in the shell is unique

$$(((S^o)[F])[L_l])\{E_l\} = (((S^o)[F])[L_l])[E_l]$$

– In the list $(((S^o)[F])[L_l])[E]$ there is at least one edge that only appears once and no edges appear more than twice; the singleton edges are on the boundary of the shell.

– The Euler condition (8), and equation (7) shall be satisfied

$$|((((S^o)[F])\{L_l^e\})\{E\})\{V\}| + |(((S^o)[F])\{L_l^v\})\{V\}| - |(((S^o)[F])\{L_l\})\{E\}| - |((S^o)[F])[L_l]| + B \text{ is even and } \leq 2 - 2|(S^o)[F]|$$

$$2 - 2H - B = |((((S^o)[F])\{L_l^e0\})\{E\})\{V\}| + |(((S^o)[F])\{L_l^v\})\{V\}| - |(((S^o)[F])\{L_l\})\{E\}| - |((S^o)[F])[L_l]| + 2|(S^o)[F]|$$

*EXPRESS* specification:

```
*)
ENTITY open_shell
  SUBTYPE OF (connected_face_set);
END_ENTITY;
(*
```

Attribute definitions:

**SELF\connected_face_set.cfs_faces:** The set of **faces**, which may include **oriented_faces**, which make up the **open_shell**.

Informal propositions:

**IP1:** Every edge shall be referenced at least once, but no more than twice by the loops of the faces.

**IP2:** Each **oriented_edge** reference shall be unique.

**IP3:** No edge may be referenced by more than two faces.

**IP4:** Distinct faces of the shell do not intersect, but may share edges, or vertices.

**IP5:** Distinct edges do not intersect, but may share vertices.

**IP6:** The Euler equation shall be satisfied.

**IP7:** The **open_shell** shall be an oriented arcwise connected 2-manifold.

**IP8:** The **open_shell** shall contain at least one hole.

**IP9:** The topological normal to each **face** of the **open_shell** shall be consistent with the topological normal to the **open_shell**.

### 5.4.24 oriented_open_shell

An **oriented_open_shell** is a **open_shell** constructed from another **open_shell** and contains a Boolean direction flag to indicate whether or not the orientation of the constructed **open_shell** agrees with the orientation of the original **open_shell**. Except for perhaps orientation, the **oriented_open_shell** is equivalent to the original **open_shell**.

*EXPRESS* specification:

```
*)
ENTITY oriented_open_shell
  SUBTYPE OF (open_shell);
  open_shell_element : open_shell;
  orientation        : BOOLEAN;
DERIVE
  SELF\connected_face_set.cfs_faces : SET [1:?] OF face
                                  := conditional_reverse(SELF.orientation,
                                       SELF.open_shell_element.cfs_faces);
WHERE
  WR1: NOT ('TOPOLOGY_SCHEMA.ORIENTED_OPEN_SHELL'
             IN TYPEOF (SELF.open_shell_element));
END_ENTITY;
(*
```

Attribute definitions:

**open_shell_element:** The open shell which defines the faces of the **oriented_open_shell**.

**orientation:** The relationship between the orientation of the **oriented_open_shell** being defined and the **open_shell_element** referenced.

**cfs_faces:** The set of faces for the **oriented_open_shell**, obtained from those of the **open_shell_element** after possibly reversing their orientation.

Formal propositions:

**WR1:** The type of **open_shell_element** shall not be an **oriented_open_shell**.

### 5.4.25 closed_shell

A **closed_shell** is a **shell** of dimensionality 2 which typically serves as a bound for a region in $R^3$. A closed shell has no boundary, and has non-zero finite extent. If the shell has a domain with coordinate space $R^3$, it divides that space into two connected regions, one finite and the other infinite. In this case, the topological normal of the shell is defined as being directed from the finite to the infinite region.

The shell is defined by a collection of **faces**, which may be **oriented_faces**. The sense of each face, after taking account of the orientation, shall agree with the shell normal as defined above. The **orientation** can be supplied directly as a Boolean attribute of an **oriented_face**, or be defaulted to TRUE if the shell member is a **face** without the orientation attribute.

The combinatorial restrictions on closed shells and geometrical restrictions on their domains ensure that any domain associated with a closed shell is a closed, orientable manifold. The domain of a closed shell, if present, is a connected, closed, oriented 2-manifold. It is always topologically equivalent to an $H$-fold torus for some $H \geq 0$. The number $H$ is referred to as the *surface genus* of the shell. If a shell of genus $H$ has a domain with coordinate space $R^3$, the finite region of space inside it is topologically equivalent to a solid ball with $H$ tunnels drilled through it.

The surface Euler equation (7) applies with $\mathcal{B} = 0$, because in this case there are no holes. As in the case of **open_shells**, the surface genus $H$ may not be known a priori, but shall be an integer $\geq 0$. Thus a necessary, but not sufficient, condition for a well-formed closed shell is the following:

$$\mathcal{V} - \mathcal{E} - \mathcal{L}_l \text{ shall be even and } \leq 2 - 2\mathcal{F} \tag{9}$$

Specifically, the following topological constraints shall be satisfied:

- Each face in the shell is unique

$$(S^c)\{F\} = (S^c)[F]$$

- Each face bound in the shell is unique

$$((S^c)[F])\{L_l\} = ((S^c)[F])[L_l]$$

- Each **oriented_edge** in the shell is unique

$$(((S^c)[F])[L_l])\{E_l\} = (((S^c)[F])[L_l])[E_l]$$

- Each edge in the shell is either used by exactly two face bounds or is used twice by one face bound

$$|(((S^c)[F])[L_l])\{E_l\}| = 2|(((S^c)[F])[L_l])\{E\}|$$

That is, in the list $(((S^c)[F])[L_l])[E]$ each edge appears exactly twice.

- The Euler conditions (9), or optionally (7) shall be satisfied

$$\begin{aligned} 2 - 2H = {} & |((((S^c)[F])\{L_l^e\})\{E\})\{V\}| + |(((S^c)[F])\{L_l^v\})\{V\}| \\ & -|(((S^c)[F])\{L_l\})\{E\}| - |((S^c)[F])[L_l]| + 2|(S^c)[F]| \end{aligned}$$

$$\begin{aligned} & |((((S^c)[F])\{L_l^e\})\{E\})\{V\}| + |(((S^c)[F])\{L_l^v\})\{V\}| - |(((S^c)[F])\{L_l\})\{E\}| \\ & \qquad - |((S^c)[F])[L_l]| \text{ is even and } \leq 2 - 2|(S^c)[F]| \end{aligned}$$

*EXPRESS* specification:

```
*)
ENTITY closed_shell
  SUBTYPE OF (connected_face_set);
END_ENTITY;
(*
```

Attribute definitions:

**SELF\connected_face_set.cfs_faces:** The set of **faces**, including **oriented faces** which define the **closed_shell**.

Informal propositions:

**IP1:** Every edge shall be referenced exactly twice by the loops of the faces.

**IP2:** Each **oriented_edge** reference shall be unique.

**IP3:** No edge shall be referenced by more than two faces.

**IP4:** Distinct faces of the shell do not intersect, but may share edges, or vertices.

**IP5:** Distinct edges do not intersect, but may share vertices.

**IP6:** Each face reference shall be unique.

**IP7:** The **loops** of the **shell** shall not be a mixture of **poly_loops** and other **loop** types.

**IP8:** The **closed_shell** shall be an oriented arcwise connected-manifold.

**IP9:** The Euler equation shall be satisfied.

**IP10:** The topological normal to each **face** of the **closed_shell** shall be consistent with the topological normal to the **closed_shell**. This implies that the topological normal to each **face**, after taking account of orientation, if present, shall point from the finite region bounded by the **closed_shell** into the infinite region outside.

### 5.4.26 oriented_closed_shell

An **oriented_closed_shell** is a **closed_shell** constructed from another **closed_shell** and contains a Boolean orientation flag to indicate whether or not the orientation of the constructed **closed_shell** agrees with the orientation of the original **closed_shell**. The **oriented_closed_shell** is equivalent to the original **closed_shell** but may have the opposite orientation..

*EXPRESS* specification:

```
*)
ENTITY oriented_closed_shell
  SUBTYPE OF (closed_shell);
  closed_shell_element : closed_shell;
  orientation          : BOOLEAN;
DERIVE
  SELF\connected_face_set.cfs_faces : SET [1:?] OF face
                                    := conditional_reverse(SELF.orientation,
                                       SELF.closed_shell_element.cfs_faces);
WHERE
  WR1: NOT ('TOPOLOGY_SCHEMA.ORIENTED_CLOSED_SHELL'
             IN TYPEOF (SELF.closed_shell_element));
```

```
END_ENTITY;
(*
```

Attribute definitions:

**closed_shell_element:** The closed shell which defines the faces of the **oriented_closed_shell**.

**orientation:** The relationship between the orientation of the **oriented_closed_shell** being defined and the **closed_shell_element** referenced.

**cfs_faces:** The set of faces for the **oriented_closed_shell**, obtained from those of the **closed_shell_element** after possibly reversing their orientation.

Formal propositions:

**WR1:** The type of **closed_shell_element** shall not be an **oriented_closed_shell**.

### 5.4.27 connected_edge_set

A **connected_edge_set** is a set of **edges** such that the domain of the edges together with their bounding vertices is arcwise connected.

*EXPRESS* specification:

```
*)
ENTITY connected_edge_set
  SUBTYPE OF (topological_representation_item);
  ces_edges : SET [1:?] OF edge;
END_ENTITY;
(*
```

Attribute definitions:

**ces_edges:** Set of **edges** arcwise connected at common **vertexs**.

Informal propositions:

**IP1:** The dimensionality of the **connected_edge_set** is 1.

**IP2:** The domains of the edges of the **connected_edge_set** shall not intersect.

## 5.5 topology_schema function definitions

### 5.5.1 conditional_reverse

Depending on its first argument, this function returns either the input topology unchanged or a copy of the input topology with its orientation reversed.

*EXPRESS* specification:

```
*)
FUNCTION conditional_reverse (p       : BOOLEAN;
                              an_item : reversible_topology)
```

```
                                                   : reversible_topology;
  IF p THEN
    RETURN (an_item);
  ELSE
    RETURN (topology_reversed (an_item));
  END_IF;
END_FUNCTION;
(*
```

Argument definitions:

**p:** (input) A Boolean value indicating whether or not orientation reversal is required.

**an_item:** (input) An item of topology which can be reversed if required.

## 5.5.2 topology_reversed

This function returns topology equivalent to the input topology except that the orientation is reversed.

*EXPRESS* specification:

```
*)
FUNCTION topology_reversed (an_item : reversible_topology)
                                     : reversible_topology;

  IF ('TOPOLOGY_SCHEMA.EDGE' IN TYPEOF (an_item)) THEN
    RETURN (edge_reversed (an_item));
  END_IF;

  IF ('TOPOLOGY_SCHEMA.PATH' IN TYPEOF (an_item)) THEN
    RETURN (path_reversed (an_item));
  END_IF;

  IF ('TOPOLOGY_SCHEMA.FACE_BOUND' IN TYPEOF (an_item)) THEN
    RETURN (face_bound_reversed (an_item));
  END_IF;

  IF ('TOPOLOGY_SCHEMA.FACE' IN TYPEOF (an_item)) THEN
    RETURN (face_reversed (an_item));
  END_IF;

  IF ('TOPOLOGY_SCHEMA.SHELL' IN TYPEOF (an_item)) THEN
    RETURN (shell_reversed (an_item));
  END_IF;

  IF ('SET' IN TYPEOF (an_item)) THEN
    RETURN (set_of_topology_reversed (an_item));
  END_IF;

  IF ('LIST' IN TYPEOF (an_item)) THEN
    RETURN (list_of_topology_reversed (an_item));
```

```
  END_IF;

  RETURN (?);
END_FUNCTION;
(*
```

Argument definitions:

**an_item:** (input) An item of reversible topology which is to have its orientation reversed.

**:** (output) A **topological_representation_item** which is the result of reversing the orientation of **an_item**.

### 5.5.3 edge_reversed

This function returns an **edge** equivalent to the input **edge** except that the orientation is reversed.

*EXPRESS* specification:

```
*)
FUNCTION edge_reversed (an_edge : edge) : edge;
  LOCAL
    the_reverse : edge;
  END_LOCAL;

  IF ('TOPOLOGY_SCHEMA.ORIENTED_EDGE' IN TYPEOF (an_edge) ) THEN
    the_reverse := oriented_edge(an_edge\oriented_edge.edge_element,
                          (NOT (an_edge\oriented_edge.orientation)));
  ELSE
    the_reverse := oriented_edge (an_edge, FALSE);
  END_IF;
  RETURN (the_reverse);
END_FUNCTION;
(*
```

Argument definitions:

**an_edge:** (input) The edge which is to have its orientation reversed.

**the_reverse:** (output) The result of the orientation reversal.

### 5.5.4 path_reversed

This function returns a **path** equivalent to the input **path** except that the orientation is reversed.

*EXPRESS* specification:

```
*)
FUNCTION path_reversed (a_path : path) : path;
  LOCAL
    the_reverse : path;
```

```
  END_LOCAL;

  IF ('TOPOLOGY_SCHEMA.ORIENTED_PATH' IN TYPEOF (a_path) ) THEN
    the_reverse := oriented_path(a_path\oriented_path.path_element,
                               (NOT(a_path\oriented_path.orientation)));
  ELSE
    the_reverse := oriented_path (a_path, FALSE);
  END_IF;

  RETURN (the_reverse);
END_FUNCTION;
(*
```

Argument definitions:

**a_path:** (input) The path which is to have its orientation reversed.

**the_reverse:** (output) The result of the orientation reversal.

### 5.5.5 face_bound_reversed

This function returns a **face_bound** equivalent to the input **face_bound** except that the orientation is reversed.

*EXPRESS* specification:

```
*)
FUNCTION face_bound_reversed (a_face_bound : face_bound) : face_bound;
  LOCAL
    the_reverse : face_bound;
  END_LOCAL;

  IF ('TOPOLOGY_SCHEMA.FACE_OUTER_BOUND' IN TYPEOF (a_face_bound) ) THEN
    the_reverse := face_bound(a_face_bound\face_bound.bound,
                          (NOT (a_face_bound\face_bound.orientation)));
  ELSE
    the_reverse := face_bound(a_face_bound.bound,
                          (NOT (a_face_bound.orientation)));
  END_IF;

  RETURN (the_reverse);
END_FUNCTION;
(*
```

Argument definitions:

**a_face_bound:** (input) The face_bound which is to have its orientation reversed.

**the_reverse:** (output) The result of the orientation reversal.

### 5.5.6 face_reversed

This function returns a **face** equivalent to input **face** except that the orientation is reversed.

*EXPRESS* specification:

```
*)
FUNCTION face_reversed (a_face : face) : face;
  LOCAL
    the_reverse : face;
  END_LOCAL;

  IF ('TOPOLOGY_SCHEMA.ORIENTED_FACE' IN TYPEOF (a_face) ) THEN
    the_reverse := oriented_face(a_face\oriented_face.face_element,
                               (NOT (a_face\oriented_face.orientation)));
  ELSE
    the_reverse := oriented_face (a_face, FALSE);
  END_IF;

  RETURN (the_reverse);
END_FUNCTION;
(*
```

Argument definitions:

**a_face:** (input) The face which is to have its orientation reversed.

**the_reverse:** (output) The result of the orientation reversal.

### 5.5.7 shell_reversed

This function returns a **shell** equivalent to the input **shell** except that the orientation is reversed.

*EXPRESS* specification:

```
*)
FUNCTION shell_reversed (a_shell : shell) : shell;
  LOCAL
    the_reverse : shell;
  END_LOCAL;

  IF ('TOPOLOGY_SCHEMA.ORIENTED_OPEN_SHELL' IN TYPEOF (a_shell) ) THEN
    the_reverse := oriented_open_shell(
                   a_shell\oriented_open_shell.open_shell_element,
                     (NOT (a_shell\oriented_open_shell.orientation)));
  ELSE
    IF ('TOPOLOGY_SCHEMA.OPEN_SHELL' IN TYPEOF (a_shell) ) THEN
      the_reverse := oriented_open_shell (a_shell, FALSE);
    ELSE
      IF ('TOPOLOGY_SCHEMA.ORIENTED_CLOSED_SHELL' IN TYPEOF (a_shell) ) THEN
        the_reverse := oriented_closed_shell(
                         a_shell\oriented_closed_shell.closed_shell_element,
                     NOT(a_shell\oriented_closed_shell.orientation));
      ELSE
```

```
        IF ('TOPOLOGY_SCHEMA.CLOSED_SHELL' IN TYPEOF (a_shell) ) THEN
          the_reverse := oriented_closed_shell (a_shell, FALSE);
        ELSE
          the_reverse := ?;
        END_IF;
      END_IF;
    END_IF;
  END_IF;

  RETURN (the_reverse);
END_FUNCTION;
(*
```

Argument definitions:

**a_shell:** (input) The shell which is to have its orientation reversed.

**the_reverse:** (output) The result of the orientation reversal.

### 5.5.8 set_of_topology_reversed

This function returns a set of topology equivalent to the input set of topology except that the orientation of each element of the set is reversed.

*EXPRESS* specification:

```
*)
FUNCTION set_of_topology_reversed (a_set : set_of_reversible_topology_item)
                                         : set_of_reversible_topology_item;
  LOCAL
    the_reverse : set_of_reversible_topology_item;
  END_LOCAL;

  the_reverse := [];
  REPEAT i := 1 TO SIZEOF (a_set);
    the_reverse := the_reverse + topology_reversed (a_set [i]);
  END_REPEAT;

  RETURN (the_reverse);
END_FUNCTION;
(*
```

Argument definitions:

**a_set:** (input) The set of topology items which are to have their orientation reversed.

**the_reverse:** (output) The result of the orientation reversal.

### 5.5.9 list_of_topology_reversed

This function returns a list of topology equivalent to the input list of topology except that the orientation of each element of the list is reversed and the order of the elements in the list is reversed.

*EXPRESS* specification:

```
*)
FUNCTION list_of_topology_reversed (a_list
                                  : list_of_reversible_topology_item)
                                  : list_of_reversible_topology_item;
  LOCAL
    the_reverse : list_of_reversible_topology_item;
  END_LOCAL;

  the_reverse := [];
  REPEAT i := 1 TO SIZEOF (a_list);
    the_reverse := topology_reversed (a_list [i]) + the_reverse;
  END_REPEAT;

  RETURN (the_reverse);
END_FUNCTION;
(*
```

Argument definitions:

**a_list:** (input) The list of topology items which are to have their orientation and list order reversed.

**the_reverse:** (output) The result of the orientation and order reversal.

### 5.5.10 boolean_choose

This function returns one of two choices depending the value of a Boolean input argument. The two choices are also input arguments.

*EXPRESS* specification:

```
*)
FUNCTION boolean_choose (b : boolean;
          choice1, choice2 : generic) : generic;

  IF b THEN
    RETURN (choice1);
  ELSE
    RETURN (choice2);
  END_IF;
END_FUNCTION;
(*
```

Argument definitions:

**b:** (input) The Boolean value used to select the element choice1 (TRUE) or choice2 (FALSE).

**choice1:** (input) The first item which may be selected.

**choice2:** (input) The second item which may be selected.

### 5.5.11 path_head_to_tail

This function returns TRUE if for the **edge**s of the input **path**, the end vertex of each **edge** is the same as the start vertex of its successor.

*EXPRESS* specification:

```
*)
FUNCTION path_head_to_tail(a_path : path) : LOGICAL;
  LOCAL
    n : INTEGER;
    p : LOGICAL := TRUE;
  END_LOCAL;

    n := SIZEOF (a_path.edge_list);
    REPEAT i := 2 TO n;
      p := p AND (a_path.edge_list[i-1].edge_end :=:
                  a_path.edge_list[i].edge_start);
    END_REPEAT;

    RETURN (p);
END_FUNCTION;
(*
```

Argument definitions:

**a_path:** (input) The path for which it is required to verify that its component edges are arranged consecutively head-to-tail.

**p:** (output) A LOGICAL variable which is TRUE if all **edge**s in the **path** join head-to-tail.

### 5.5.12 list_face_loops

Given a **face** (or a **subface**), the function returns the list of **loop**s in the **face** or **subface**.

*EXPRESS* specification:

```
*)
FUNCTION list_face_loops(f: face) : LIST[0:?] OF loop;
  LOCAL
    loops : LIST[0:?] OF loop := [];
  END_LOCAL;

  REPEAT i := 1 TO SIZEOF(f.bounds);
    loops := loops +(f.bounds[i].bound);
  END_REPEAT;
```

```
  RETURN(loops);
END_FUNCTION;
(*
```

Argument definitions:

**f:** (input) The **face** for which it is required to generate the list of bounding **loops**.

**loops:** (output) The list of **loops** for **f**.

## 5.5.13 list_loop_edges

Given a **loop**, the function returns the list of **edges** in the **loop**.

*EXPRESS* specification:

```
*)
FUNCTION list_loop_edges(l: loop): LIST[0:?] OF edge;
  LOCAL
    edges : LIST[0:?] OF edge := [];
  END_LOCAL;

  IF 'TOPOLOGY_SCHEMA.EDGE_LOOP' IN TYPEOF(l) THEN
    REPEAT i := 1 TO SIZEOF(l\path.edge_list);
      edges := edges + (l\path.edge_list[i].edge_element);
    END_REPEAT;
  END_IF;

  RETURN(edges);
END_FUNCTION;
(*
```

Argument definitions:

**l:** (input) The **loop** for which it is required to generate the list of **edges**.

**edges:** (output) The list of **edges** for **l**.

## 5.5.14 list_shell_edges

Given a **shell**, the function returns the list of **edges** in the **shell**.

*EXPRESS* specification:

```
*)
FUNCTION list_shell_edges(s : shell) : LIST[0:?] OF edge;
  LOCAL
    edges : LIST[0:?] OF edge := [];
  END_LOCAL;

  REPEAT i := 1 TO SIZEOF(list_shell_loops(s));
```

```
    edges := edges + list_loop_edges(list_shell_loops(s)[i]);
  END_REPEAT;

  RETURN(edges);
END_FUNCTION;
(*
```

Argument definitions:

s: (input) The **shell** for which it is required to generate the list of **edges**.

**edges:** (output) The list of **edges** for s.

### 5.5.15 list_shell_faces

Given a **shell**, the function returns the list of **faces** in the **shell**.

*EXPRESS* specification:

```
*)
FUNCTION list_shell_faces(s : shell) : LIST[0:?] OF face;
  LOCAL
    faces : LIST[0:?] OF face := [];
  END_LOCAL;

  IF ('TOPOLOGY_SCHEMA.CLOSED_SHELL' IN TYPEOF(s)) OR
     ('TOPOLOGY_SCHEMA.OPEN_SHELL' IN TYPEOF(s)) THEN
    REPEAT i := 1 TO SIZEOF(s\connected_face_set.cfs_faces);
      faces := faces + s\connected_face_set.cfs_faces[i];
    END_REPEAT;
  END_IF;

  RETURN(faces);
END_FUNCTION;
(*
```

Argument definitions:

s: (input) The shell for which it is required to generate the list of faces.

**faces:** (output) The list of faces for s.

### 5.5.16 list_shell_loops

Given a **shell**, the function returns the list of **loops** in the **shell**.

*EXPRESS* specification:

```
*)
FUNCTION list_shell_loops(s : shell) : LIST[0:?] OF loop;
  LOCAL
    loops : LIST[0:?] OF loop := [];
```

```
  END_LOCAL;

  IF 'TOPOLOGY_SCHEMA.VERTEX_SHELL' IN TYPEOF(s) THEN
    loops := loops + s.vertex_shell_extent;
  END_IF;

  IF 'TOPOLOGY_SCHEMA.WIRE_SHELL' IN TYPEOF(s) THEN
    REPEAT i := 1 TO SIZEOF(s.wire_shell_extent);
      loops := loops + s.wire_shell_extent[i];
    END_REPEAT;
  END_IF;

  IF ('TOPOLOGY_SCHEMA.OPEN_SHELL' IN TYPEOF(s)) OR
     ('TOPOLOGY_SCHEMA.CLOSED_SHELL' IN TYPEOF(s)) THEN
    REPEAT i := 1 TO SIZEOF(s.cfs_faces);
      loops := loops + list_face_loops(s.cfs_faces[i]);
    END_REPEAT;
  END_IF;

  RETURN(loops);
END_FUNCTION;
(*
```

Argument definitions:

s: (input) The **shell** for which it is required to generate the list of **loops**.

**loops:** (output) The list of **loops** for s.

## 5.5.17 mixed_loop_type_set

Given a set of **loops**, the function returns TRUE if the set includes both **poly_loops** and other types (edge and vertex) of loops.

*EXPRESS* specification:

```
*)
 FUNCTION mixed_loop_type_set(l: SET[0:?] OF loop): LOGICAL;
   LOCAL
     i            : INTEGER;
     poly_loop_type: LOGICAL;
   END_LOCAL;
   IF(SIZEOF(l) <= 1) THEN
     RETURN(FALSE);
   END_IF;
   poly_loop_type := ('TOPOLOGY_SCHEMA.POLY_LOOP' IN TYPEOF(l[1]));
   REPEAT i := 2 TO SIZEOF(l);
     IF(('TOPOLOGY_SCHEMA.POLY_LOOP' IN TYPEOF(l[i])) <> poly_loop_type) THEN
         RETURN(TRUE);
      END_IF;
   END_REPEAT;
   RETURN(FALSE);
```

```
 END_FUNCTION;
(*
```

Argument definitions:

**l:** (input) The set of **loops** for which it is required to determine whether, or not, it is a mixture of **poly_loops** and others.

### 5.5.18 list_to_set

This function creates a **SET** from a **LIST**, the type of element for the **SET** will be the same as that in the original **LIST**.

*EXPRESS* specification:

```
*)
FUNCTION list_to_set(l : LIST [0:?] OF GENERIC:T) : SET OF GENERIC:T;
  LOCAL
    s : SET OF GENERIC:T := [];
  END_LOCAL;

  REPEAT i := 1 TO SIZEOF(l);
    s := s + l[i];
  END_REPEAT;

  RETURN(s);
END_FUNCTION;
(*
```

Argument definitions:

**l:** (input) The list of elements to be converted to a set.

**s:** (output) The set corresponding to **l**.



### 5.5.19 edge_curve_pcurves

This function returns the set of pcurves that are associated with (i.e., represent the geometry of) an **edge_curve**.

*EXPRESS* specification:

```
*)
FUNCTION edge_curve_pcurves (an_edge   : edge_curve;
                             the_surface_curves : SET OF surface_curve)
      : SET OF pcurve;
LOCAL
  a_curve      : curve;
  result       : SET OF pcurve;
  the_geometry : LIST[1:2] OF pcurve_or_surface;
END_LOCAL;
  a_curve := an_edge.edge_geometry;
```

```
  result := [];
  IF 'GEOMETRY_SCHEMA.PCURVE' IN TYPEOF(a_curve) THEN
    result := result + a_curve;
  ELSE
    IF 'GEOMETRY_SCHEMA.SURFACE_CURVE' IN TYPEOF(a_curve) THEN
      the_geometry := a_curve\surface_curve.associated_geometry;
      REPEAT k := 1 TO SIZEOF(the_geometry);
         IF 'GEOMETRY_SCHEMA.PCURVE' IN TYPEOF (the_geometry[k])
         THEN
            result := result + the_geometry[k];
         END_IF;
      END_REPEAT;
    ELSE
      REPEAT j := 1 TO SIZEOF(the_surface_curves);
        the_geometry := the_surface_curves[j].associated_geometry;
        IF the_surface_curves[j].curve_3d :=: a_curve
        THEN
          REPEAT k := 1 TO SIZEOF(the_geometry);
            IF 'GEOMETRY_SCHEMA.PCURVE' IN TYPEOF (the_geometry[k])
            THEN
              result := result + the_geometry[k];
            END_IF;
          END_REPEAT;
        END_IF;
      END_REPEAT;
    END_IF;
  END_IF;

  RETURN (RESULT);
END_FUNCTION;
```

(*

Argument definitions:

**an_edge:** (input) The **edge_curve** whose associated pcurves are to be found.

**the_surface_curves:** (input) The set of all **surface_curve**s within the scope of the search for **pcurve**s.

**result:** (output) The set of all **pcurve**s associated with **an_edge**.



## 5.5.20 vertex_point_pcurves

This function returns the set of **pcurve**s that are associated with (i.e., represent the geometry of) a **vertex_point**.

*EXPRESS* specification:

*)

```
FUNCTION vertex_point_pcurves (a_vertex  : vertex_point;
      the_degenerates : SET OF evaluated_degenerate_pcurve)
      : SET OF degenerate_pcurve;
```

```
LOCAL
  a_point : point;
  result  : SET OF degenerate_pcurve;
END_LOCAL;
  a_point := a_vertex.vertex_geometry;
  result := [];
  IF 'GEOMETRY_SCHEMA.DEGENERATE_PCURVE' IN TYPEOF(a_point) THEN
    result := result + a_point;
  ELSE
      REPEAT j := 1 TO SIZEOF(the_degenerates);
         IF (the_degenerates[j].equivalent_point :=: a_point)  THEN
            result := result + the_degenerates[j];
         END_IF;
      END_REPEAT;
  END_IF;

  RETURN (RESULT);
END_FUNCTION;

(*
```

Argument definitions:

**a_vertex:** (input) The **vertex_point** whose associated pcurves are to be found.

**the_degenerates:** (input) The set of all **evaluated_degenerate_pcurves** within the scope of the search for **pcurves**.

**result:** (output) The set of all **degenerate_pcurves** having the same geometry as **a_vertex**.

*EXPRESS* specification:

```
*)
END_SCHEMA; -- end TOPOLOGY schema
(*
```

# 6 Geometric models

The following EXPRESS declaration begins the **geometric_model_schema** and identifies the necessary external references.

*EXPRESS* specification:

```
*)
SCHEMA geometric_model_schema;
  REFERENCE FROM geometry_schema;
  REFERENCE FROM topology_schema;
  REFERENCE FROM measure_schema(length_measure,
                               positive_length_measure,
                               plane_angle_measure,
                               plane_angle_unit,
                               positive_plane_angle_measure);
(*
```

NOTES

1 – The schemas referenced above can be found in the following Parts of ISO 10303:

| | |
|---|---|
| geometry_schema | Clause 4 of this part of ISO 10303 |
| topology_schema | Clause 5 of this part of ISO 10303 |
| measure_schema | ISO 10303-41 |

2 – See annex D, figures D.16 - D.18, for a graphical presentation of this schema.



## 6.1 Introduction

The subject of the **geometric_model** schema is the set of basic resources necessary for the communication of data describing the size, position, and shape of objects. The **solid_model** subtypes provide basic resources for the communication of data describing the precise size and shape of three-dimensional solid objects. The two classical types of solid model, constructive solid geometry (CSG) and boundary representation (B-rep) are included. Also included in this clause are entities providing less complete geometric and topological information than the full CSG or B-rep models. The use of these entities is appropriate for communication with systems whose capability differs from that of solid modelling systems.

The entities in this schema are arranged in a logical order beginning with the **solid_model** supertype and its various subtypes. These subtypes include the different types of boundary representations (B-reps) and the CSG solids. After the **solid_model** subtypes the surface model entities are grouped together, followed by the wireframe models and the geometric sets.

## 6.2 Fundamental concepts and assumptions

The constructive solid geometry models are represented by their component primitives and the sequence of Boolean operations (`union`, `intersection` or `difference`) used in their construction. The standard CSG primitives are the **cone**, **cylinder**, **sphere**, **torus**, **block** and **right_angular_wedge** and should be defined in their final position and orientation. The entity which communicates the logical sequence of Boolean operations is the **boolean_result** which identifies an operator and two operands. The operands can themselves be Boolean results, thus enabling nested operations. In addition to the CSG primitives, any solid model, including, in particular, swept solids and half-space solids may be Boolean operands. The swept solids are the **solid_of_revolution** and the **solid_of_linear_extrusion**. The swept solids are obtained by extruding or sweeping a planar face which may contain holes. The **half_space_solid** is essentially defined as a semi-infinite solid on one side of a surface; it may be limited by a **box_domain**.

B-rep models are represented by the set of shells defining their exterior or interior boundaries. Constraints ensure that the associated geometry is well defined and that the Euler formula connecting the numbers of vertices, edges, faces, loops and shells in the model is satisfied. The **faceted_brep** is restricted to represent B-reps in which all faces are planar and every loop is a **poly_loop**.

The **solid_replica** entity provides a mechanism for copying an existing solid in a new location.

The **shell_based_surface_model**, **face_based_surface_model**, **shell_based_wireframe_model**, **edge_based_wireframe_model**, **geometric_set**, and **geometric_curve_set** entities do not enforce the integrity checks of the **manifold_solid_brep** and can be used for the communication of incomplete models or non-manifold objects, including two-dimensional models.

## 6.3 geometric_model_schema type definitions

### 6.3.1 boolean_operand

This select type identifies all those types of entities which may participate in a boolean operation to form a CSG solid.

*EXPRESS* specification:

```
*)
TYPE boolean_operand = SELECT
  (solid_model,
   half_space_solid,
   csg_primitive,
   boolean_result);
END_TYPE;
(*
```

### 6.3.2 boolean_operator

This type defines the three boolean operators used in the definition of CSG solids.

*EXPRESS* specification:

```
*)
TYPE boolean_operator = ENUMERATION OF
  (union,
   intersection,
   difference);
END_TYPE;
(*
```

**Definitions:**

**union:** The operation of constructing the regularised set theoretic union of the volumes defined by two solids.

**intersection:** The operation of constructing the regularised set theoretic intersection of the volumes defined by two solids.

**difference:** The regularised set theoretic difference between the volumes defined by two solids.

### 6.3.3 csg_primitive

This select type defines the set of CSG primitives which may participate in boolean operations. The CSG primitives are **sphere, right_circular_cone, right_circular_cylinder, torus, block** and
**right_angular_wedge**.

*EXPRESS* specification:

```
*)
TYPE csg_primitive = SELECT
  (sphere,
   block,
   right_angular_wedge,
   torus,
   right_circular_cone,
   right_circular_cylinder);
END_TYPE;
(*
```



### 6.3.4 csg_select

This type identifies the types of entity which may be selected as the root of a CSG tree including a single CSG primitive as a special case.

*EXPRESS* specification:

```
*)
TYPE csg_select = SELECT
  (boolean_result,
```

```
   csg_primitive);
END_TYPE;
(*
```

### 6.3.5 geometric_set_select

This select type identifies the types of entities which can occur in a **geometric_set**.

*EXPRESS* specification:

```
*)
TYPE geometric_set_select = SELECT
  (point,
   curve,
   surface);
END_TYPE;
(*
```

### 6.3.6 surface_model

This type collects all possible surface model entities.

Some product model representations consist of collections of surfaces which do not necessarily form the complete boundary of a solid. Such a model can be represented by a collection of **faces** or **shells**.

*EXPRESS* specification:

```
*)
TYPE surface_model = SELECT
  (shell_based_surface_model,
   face_based_surface_model);
END_TYPE;
(*
```

### 6.3.7 wireframe_model

This type collects all possible wireframe model entities.

A wireframe representation of a geometric model contains information only about the intersections of the surfaces forming the boundary but does not contain information about the surfaces themselves.

*EXPRESS* specification:

```
*)
TYPE wireframe_model = SELECT
  (shell_based_wireframe_model,
   edge_based_wireframe_model);
END_TYPE;
(*
```

## 6.4 geometric_model_schema entity definitions

The following entities are used in the **geometric_model_schema**.

### 6.4.1 solid_model

A **solid_model** is a complete representation of the nominal shape of a product such that all points in the interior are connected. Any point can be classified as being inside, outside or on the boundary of a solid.

There are several different types of solid model representations.

*EXPRESS* specification:

```
*)
ENTITY solid_model
  SUPERTYPE OF (ONEOF( csg_solid, manifold_solid_brep, swept_face_solid,
                       swept_area_solid, solid_replica))
  SUBTYPE OF (geometric_representation_item);
END_ENTITY;
(*
```



### 6.4.2 manifold_solid_brep

A **manifold_solid_brep** is a finite, arcwise connected volume bounded by one or more surfaces, each of which is a connected, oriented, finite, closed 2-manifold. There is no restriction on the number of through holes, nor on the number of voids within the volume.

The Boundary Representation (B-rep) of a manifold solid utilises a graph of edges and vertices embedded in a connected, oriented, finite, closed two manifold surface. The embedded graph divides the surface into arcwise connected areas known as faces. The edges and vertices, therefore, form the boundaries of the faces and the domain of a face does not include its boundaries. The embedded graph may be disconnected and may be a pseudograph. The graph is labelled; that is, each entity in the graph has a unique identity. The geometric surface definition used to specify the geometry of a face shall be 2-manifold embeddable in the plane within the domain of the face. In other words, it shall be connected, oriented, finite, non-self-intersecting, and of surface genus 0.

Faces do not intersect except along their boundaries. Each edge along the boundary of a face is shared by at most one other face in the assemblage. The assemblage of edges in the B-rep do not intersect except at their boundaries (i.e., vertices). The geometric curve definition used to specify the geometry of an edge shall be arcwise connected and shall not self intersect or overlap within the domain of the edge. The geometry of an edge shall be consistent with the geometry of the faces of which it forms a partial bound.

The geometry used to define a vertex shall be consistent with the geometry of the faces and edges of which it forms a partial bound.

A B-rep is represented by one or more **closed_shell**s which shall be disjoint. One shell, the outer, shall completely enclose all the other shells and no other shell may enclose a shell. The facility to define a B-rep with one or more internal voids is provided by the **brep_with_voids** subtype. The following version of the Euler formula shall be satisfied

$$\chi_{ms} = \mathcal{V} - \mathcal{E} + 2\mathcal{F} - \mathcal{L}_l - 2(\mathcal{S} - \mathcal{G}^s) = 0 \tag{10}$$

where $\mathcal{V}, \mathcal{E}, \mathcal{F}, \mathcal{L}_l$ and $\mathcal{S}$ are the numbers of unique vertices, edges, faces, face bounds and shells in the model and $\mathcal{G}^s$ is the sum of the genus of the shells.

More specifically, the topological entities shall conform to the following constraints, where $B$ denotes a manifold solid B-rep:

– The shells shall be unique

$$(B)[S] = (B)\{S\}$$

– Each face in the B-rep is unique

$$((B)[S])[F] = ((B)[S])\{F\}$$

– Each loop is unique

$$(((B)[S])[F])[L] = (((B)[S])[F])\{L\}$$

– Each (edge + logical) pair is unique

$$((((B)[S])[F])[L])[E_l] = ((((B)[S])[F])[L])\{E_l\}$$

– Each edge in the B-rep is either used by exactly two loops or twice by one loop

$$|((((B)[S])[F])[L])\{E_l\}| = 2|((((B)[S])[F])[L])[E_l]|$$

That is, in the list $((((B)[S])[F])[L])[E]$ each edge appears exactly twice.

– Equation (10) shall be satisfied

$$\begin{aligned} 2|(B)[S]| - 2\sum G^s &= |(((((B)[S])[F])\{L^e\})\{E\})\{V\}| + |((((B)[S])[F])\{L^v\})\{V\}| \\ &\quad -|((((B)[S])[F])\{L\})\{E\}| + 2|((B)[S])[F]| - |(((B)[S])[F])[L]| \end{aligned}$$

The topological normal of the B-rep at each point on its boundary is the surface normal direction that points away from the solid material. The **closed_shell** normals, as used, shall be consistent with the topological normal of the B-rep. The **manifold_solid_brep** has two subtypes, **faceted_brep** and **brep_with_voids**, with which there exists a default ANDOR relationship. The following can all be instantiated:

– **manifold_solid_brep**

- **brep_with_voids**
- **faceted_brep**
- **faceted_brep** AND **brep_with_voids**

*EXPRESS* specification:

```
*)
ENTITY manifold_solid_brep
  SUBTYPE OF (solid_model);
  outer : closed_shell;
END_ENTITY;
(*
```

Attribute definitions:

**outer:** A **closed_shell** defining the exterior boundary of the solid. The shell normal shall point away from the interior of the solid.

Informal propositions:

**IP1:** The dimensionality of a **manifold_solid_brep** shall be 3.

**IP2:** The extent of the **manifold_solid_brep** shall be finite and non-zero.

**IP3:** No **vertex_point**, undirected **edge_curve** (i.e, one which is not a **oriented_edge**), or undirected **face_surface** (i.e., one which is not a **oriented_face**) referenced by a **manifold_solid_brep** shall intersect any other **vertex_point**, undirected **edge_curve**, or undirected **face_surface** referenced by the same **manifold_solid_brep**.

**IP4:** Distinct **loop**s referenced by the same **face** shall have no common **vertex**s.

NOTE – This implies that distinct loops of the same face have no common edges. If geometry is present, distinct loops of the same face do not intersect.

**IP5:** All topological elements of the **manifold_solid_brep** shall have defined associated geometry.

**IP6:** The shell normals shall agree with the B-rep normal and point away from the solid represented by the B-rep.

**IP7:** Each face shall be referenced only once by the shells of the **manifold_solid_brep**.

**IP8:** Each **oriented_edge** in the **manifold_solid_brep** shall be referenced only once.

**IP9:** Each undirected edge shall be referenced exactly twice by the loops in the faces of the **manifold_solid_brep**'s shells.

**IP10:** The Euler equation shall be satisfied for the boundary representation, where the genus term **shell_genus** is the sum of the genus values for the shells of the brep.

**IP11:** A **manifold_solid_brep**, which is not a faceted brep, shall not reference **poly_loops**.

**IP12:** A **faceted_brep** can reference only **poly_loops** as face boundaries.

### 6.4.3 brep_with_voids

A **brep_with_voids** is a special subtype of the **manifold_solid_brep** which contains one or more voids in its interior. The voids are represented by **oriented_closed_shells** which are defined so that the **oriented_closed_shell** normals point into the void, that is, with **orientation** FALSE. A **brep_with_voids** can also be a **faceted_brep**.

*EXPRESS* specification:

```
*)
ENTITY brep_with_voids
  SUBTYPE OF (manifold_solid_brep);
  voids : SET [1:?] OF oriented_closed_shell;
END_ENTITY;
(*
```

Attribute definitions:

**SELF\manifold_solid_brep.outer:** An **oriented_closed_shell** defining the exterior boundary of the solid. The shell normal shall point away from the interior of the solid.

**voids:** Set of **oriented_closed_shells** defining voids within the solid. The set may contain one or more shells.

Informal propositions:

**IP1:** Each void shell shall be disjoint from the outer shell and from every other void shell.

**IP2:** Each void shell shall be enclosed within the outer shell but not within any other void shell. In particular, the outer shell is not in the set of void shells.

**IP3:** Each shell in the **manifold_solid_brep** shall be referenced only once.

### 6.4.4 faceted_brep

A **faceted_brep** is a simple form of boundary representation model in which all faces are planar and all edges are straight lines.

NOTE – The **faceted_brep** has been introduced in order to support the large number of systems that allow boundary type solid representations with planar surfaces only. Faceted models may be represented by **manifold_solid_brep** but their representation as a **faceted_brep** will be more compact.

Unlike the B-rep model, edges and vertices are not represented explicitly in the model but are implicitly available through the **poly_loop** entity. A **faceted_brep** has to meet the same topological constraints as the **manifold_solid_brep**.

*EXPRESS* specification:

```
*)
ENTITY faceted_brep
  SUBTYPE OF (manifold_solid_brep);
END_ENTITY;
(*
```

Informal propositions:

**IP1:** All the bounding loops of all the faces of all the shells in the **faceted_brep** shall be of type **poly_loop**.

**IP2:** The faces in the shells may have implicit or explicit surface geometry. If explicit, the face surface shall be a plane. All polyloops defining the face shall be coplanar.

### 6.4.5 csg_solid

A solid represented as a CSG model is defined by a collection of so-called primitive solids, combined using regularised boolean operations. The allowed operations are intersection, union and difference. As a special case a CSG solid can also consist of a single CSG primitive.

A regularised subset of space is the closure of its interior, where this phrase is interpreted in the usual sense of point set topology. For **boolean_results** regularisation has the effect of removing dangling edges and other anomalies produced by the original operations.

A CSG solid requires two kinds of information for its complete definition: geometric and structural.

The geometric information is conveyed by **solid_models**. These typically are primitive volumes such as cylinders, wedges and extrusions, but can include general brep models. Solid_models can also be **solid_replicas** (transformed solids) and **half_space_solids**.

The structural information is in a tree (strictly, an acyclic directed graph) of **boolean_result** and CSG solids, which represent a 'recipe' for building the solid. The terminal nodes are the geometric primitives and other solids. Every **csg_solid** has precisely one **boolean_result** associated with it which is the root of the tree that defines the solid. (There may be further **boolean_results** within the tree as operands). The significance of a **csg_solid** entity is that the solid defined by the associated tree is thus identified as a significant object in itself, and in this way it is distinguished from other **boolean_result** entities representing intermediate results during the construction process.

*EXPRESS* specification:

```
*)
ENTITY csg_solid
  SUBTYPE OF (solid_model);
  tree_root_expression : csg_select;
END_ENTITY;
(*
```

Attribute definitions:

**tree_root_expression:** Boolean expression of primitives and regularised operators describing the solid. The root of the tree of boolean expressions is given here explicitly as a **boolean_result** entity, or as a **csg_primitive**.

### 6.4.6 boolean_result

A **boolean_result** is the result of a regularised operation on two solids to create a new solid. Valid operations are regularised union, regularised intersection, and regularised difference. For purposes of Boolean operations, a solid is considered to be a regularised set of points.

The final **boolean_result** depends upon the operation and the two operands. In the case of the difference operator the order of the operands is also significant. The operator can be either **union**, **intersection** or **difference**. The effect of these operators is described below.

**Union** on two solids is the new solid that contains all the points that are in either the **first_operand** or the **second_operand** or both.

**Intersection** on two solids is the new solid that is the regularisation of the set of all points that are in both the **first_operand** and the **second_operand**.

The result of the **difference** operation on two solids is the regularisation of the set of all points which are in the **first_operand**, but not in the **second_operand**.

NOTE – For example if the first operand is a block and the second operand is a solid cylinder of suitable dimensions and location the **boolean_result** produced with the difference operator would be a block with a circular hole.

*EXPRESS* specification:

```
*)
ENTITY boolean_result
  SUBTYPE OF (geometric_representation_item);
  operator        : boolean_operator;
  first_operand   :  boolean_operand;
  second_operand  :  boolean_operand;
END_ENTITY;
(*
```

Attribute definitions:

**operator:** The boolean operator used in the operation to create the result.

**first_operand:** The first operand to be operated upon by the boolean operation.

**second_operand:** The second operand specified for the operation.

### 6.4.7 sphere

A **sphere** is a CSG primitive with a spherical shape defined by a centre and a radius.

*EXPRESS* specification:

```
*)
ENTITY sphere
  SUBTYPE OF (geometric_representation_item);
  radius : positive_length_measure;
  centre : point;
END_ENTITY;
(*
```

Attribute definitions:

**radius:** The radius of the **sphere**.

**centre:** The location of the centre of the **sphere**.

### 6.4.8 right_circular_cone

A **right_circular_cone** is a CSG primitive in the form of a cone which may be truncated. It is defined by an axis, a point on the axis, the semi-angle of the cone, and a distance giving the location in the negative direction along the axis from the point to the base of the cone. In addition, a radius is given, which, if nonzero, gives the size and location of a truncated face of the cone.

*EXPRESS* specification:

```
*)
ENTITY right_circular_cone
  SUBTYPE OF (geometric_representation_item);
  position   : axis1_placement;
  height     : positive_length_measure;
  radius     : length_measure;
  semi_angle : plane_angle_measure;
WHERE
  WR1: radius >= 0.0;
END_ENTITY;
(*
```

Attribute definitions:

**position:** The location of a **point** on the axis and the direction of the axis.

**position.location:** A **point** on the axis of the cone and on one of the planar circular faces, or, if radius is zero, at the apex.

**position.axis:** The direction of the central axis of symmetry of the cone.

**height:** The distance between the planar circular faces of the cone, if **radius** is greater than zero; or from the base to the apex, if radius equals zero.

**radius:** The radius of the cone at the point on the axis (**position.location**). If the **radius** is zero, the cone has an apex at this point. If the **radius** is greater than zero, the cone is truncated.

**semi_angle:** One half the angle of the cone. This is the angle between the axis and a generator of the conical surface.

Formal propositions:

**WR1:** The **radius** shall be non-negative.

Informal propositions:

**IP1:** The **semi_angle** shall be between 0° and 90°.

### 6.4.9 right_circular_cylinder

A **right_circular_cylinder** is a CSG primitive in the form of a solid cylinder of finite height. It is defined by an axis point at the centre of one planar circular face, an axis, a height, and a radius. The faces are perpendicular to the axis and are circular discs with the specified radius. The height is the distance from the first circular face centre in the positive direction of the axis to the second circular face centre.

*EXPRESS* specification:

```
*)

ENTITY right_circular_cylinder
  SUBTYPE OF (geometric_representation_item);
  position    : axis1_placement;
  height      : positive_length_measure;
  radius      : positive_length_measure;
END_ENTITY;
(*
```

Attribute definitions:

**position:** The location of a **point** on the axis and the direction of the axis.

**position.location:** A **point** on the axis of the cylinder and at the centre of one of the planar circular faces.

**position.axis:** The direction of the central axis of symmetry of the cylinder.

**height:** The distance between the planar circular faces of the cylinder.

**radius:** The radius of the cylinder.

### 6.4.10 torus

A **torus** is a solid primitive defined by sweeping the area of a circle (the generatrix) about a larger circle (the directrix). The directrix is defined by a location and direction (**axis1_placement**).

*EXPRESS* specification:

```
*)
ENTITY torus
```

```
  SUBTYPE OF (geometric_representation_item);
  position     : axis1_placement;
  major_radius : positive_length_measure;
  minor_radius : positive_length_measure;
WHERE
  WR1: major_radius > minor_radius;
END_ENTITY;
(*
```

Attribute definitions:

**position:** The location of the central **point** on the axis and the direction of the axis. This defines the centre and plane of the directrix.

**major_radius:** The radius of the directrix.

**minor_radius:** The radius of the generatrix.

Formal propositions:

**WR1:** The **major_radius** shall be greater than the **minor_radius**.

## 6.4.11 block

A **block** is a solid rectangular parallelepiped, defined with a location and placement coordinate system. The **block** is specified by the positive lengths **x**, **y**, and **z** along the axes of the placement coordinate system, and has one vertex at the origin of the placement coordinate system.

*EXPRESS* specification:

```
*)
ENTITY block
  SUBTYPE OF (geometric_representation_item);
  position : axis2_placement_3d;
  x        : positive_length_measure;
  y        : positive_length_measure;
  z        : positive_length_measure;
END_ENTITY;
(*
```

Attribute definitions:

**position:** The location and orientation of the axis system for the primitive. The block has one vertex at **position.location** and its edges aligned with the placement axes in the positive sense.

**x:** The size of the block along the placement X axis, (position.p[1]).

**y:** The size of the block along the placement Y axis, (position.p[2]).

**z:** The size of the block along the placement Z axis, (position.p[3]).

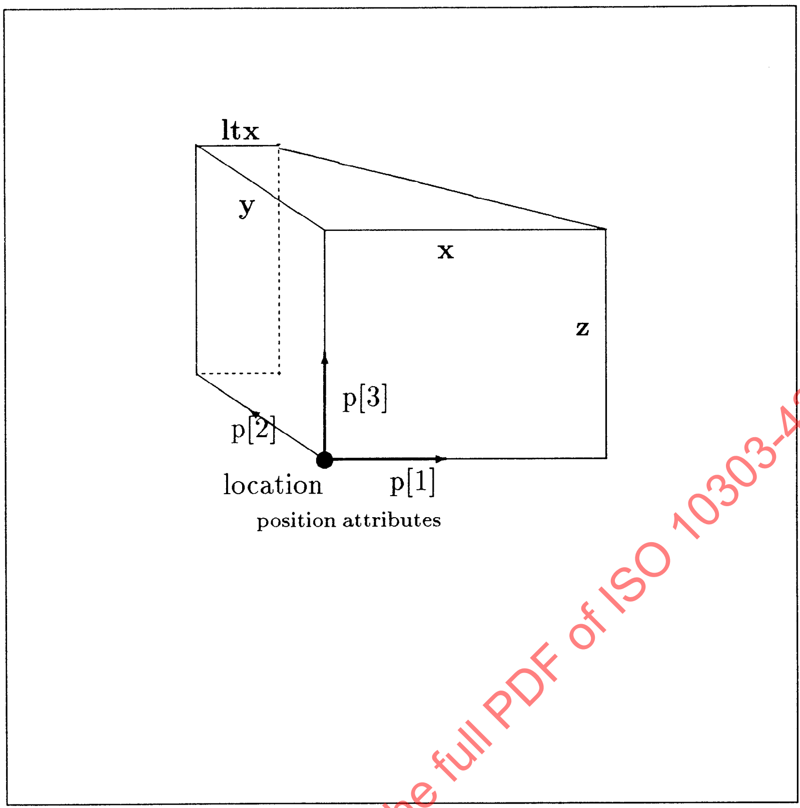




**Figure 12 – Right angular wedge and its attributes**

### 6.4.12 right_angular_wedge

A **right_angular_wedge** can be envisioned as the result of intersecting a block with a plane perpendicular to one of its faces. It is defined with a location and local coordinate system. A triangular/trapezoidal face lies in the plane defined by the placement X and Y axes. This face is defined by positive lengths **x** and **y** along the placement X and Y axes, by the length **ltx** (if nonzero) parallel to the X axis at a distance **y** from the placement origin, and by the line connecting the ends of the **x** and **ltx** segments. The remainder of the wedge is specified by the positive length **z** along the placement Z axis which defines a distance through which the trapezoid or triangle is extruded. If LTX = 0, the wedge has five faces; otherwise, it has six faces.

NOTE – See figure 12 for interpretation of attributes.

*EXPRESS* specification:

```
*)
ENTITY right_angular_wedge
 SUBTYPE OF (geometric_representation_item);
 position : axis2_placement_3d;
 x        : positive_length_measure;
```

```
  y        : positive_length_measure;
  z        : positive_length_measure;
  ltx      : length_measure;
WHERE
  WR1: ((0.0 <= ltx) AND (ltx < x));
END_ENTITY;
(*
```

Attribute definitions:

**position:** The location and orientation of the placement axis system for the primitive. The wedge has one vertex at **position.location** and its edges aligned with the placement axes in the positive sense.

**x:** The size of the wedge along the placement X axis.

**y:** The size of the wedge along the placement Y axis.

**z:** The size of the wedge along the placement Z axis.

**ltx:** The length in the positive X direction of the smaller surface of the wedge.

Formal propositions:

**WR1: ltx** shall be non-negative and less than **x**.

### 6.4.13 swept_face_solid

The **swept_area_solid** entity collects the entities which are defined procedurally by a sweeping action on planar figures. The position in space of the swept solid will be dependent upon the position of the **swept_face**. The **swept_face** will be a face of the swept area solid, except for the case of a solid of revolution with angle equal to 360 degrees.

*EXPRESS* specification:

```
*)
ENTITY swept_face_solid
  SUPERTYPE OF (ONEOF(extruded_face_solid, revolved_face_solid))
  SUBTYPE OF (solid_model);
  swept_face :  face_surface;
WHERE
  WR1: 'GEOMETRY_SCHEMA.PLANE' IN TYPEOF(swept_face.face_geometry);
END_ENTITY;
(*
```

Attribute definitions:

**swept_face:** The **face_surface** defining the area to be swept. The extent of this face is defined by the **bounds** attribute of the referenced **face_surface**.

Formal propositions:

**WR1:** The **swept_face** shall be planar. The **face_geometry** attribute of the **face_surface** referenced shall be a **plane**.

### 6.4.14 extruded_face_solid

An **extruded_face_solid** is a solid defined by sweeping a planar **face**. The direction of translation is defined by a **direction** vector, and the length of the translation is defined by a distance **depth**. The planar face may have holes which will sweep into holes in the solid.

*EXPRESS* specification:

```
*)
ENTITY extruded_face_solid
  SUBTYPE OF (swept_face_solid);
  extruded_direction : direction;
  depth              : positive_length_measure;
WHERE
  WR1: dot_product(
       (SELF\swept_face_solid.swept_face.face_geometry\
       elementary_surface.position.p[3])
                                     , extruded_direction) <> 0.0;
END_ENTITY;
(*
```

Attribute definitions:

**SELF\swept_face_solid.swept_face:** The face to be extruded to produce the solid.

**extruded_direction:** The **direction** in which the face is to be swept.

**depth:** The distance the face is to be swept.

Formal propositions:

**WR1:** **extruded_direction** shall not be perpendicular to the normal to the plane of the **extruded_face**.

### 6.4.15 revolved_face_solid

A **revolved_face_solid** is a solid of revolution formed by revolving a planar **face** about an axis. The axis shall be in the plane of the face and the axis shall not intersect the interior of the face. The planar face may have holes which will sweep into holes in the solid. The direction of revolution is clockwise when viewed along the axis in the positive direction. More precisely if **A** is the axis location and **d** is the axis direction and **C** is an arc on the surface of revolution generated by an arbitrary point **p** on the boundary of the face, then **C** leaves **p** in direction **d** × (**p**-**A**) as the face is revolved.

NOTE – See figure 13 for illustration of attributes.

*EXPRESS* specification:

```
*)
ENTITY revolved_face_solid
```