



Open CASCADE Technology  
6.8.0

Boolean Operations

November 7, 2014

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Overview</b>	<b>2</b>
2.1	Operators	2
2.1.1	Boolean operator	2
2.1.2	General Fuse operator	2
2.1.3	Partition operator	3
2.2	Parts of algorithms	3
<b>3</b>	<b>Terms and Definitions</b>	<b>4</b>
3.1	Interferences	4
3.1.1	Vertex/Vertex interference	4
3.1.2	Vertex/Edge interference	4
3.1.3	Vertex/Face interference	5
3.1.4	Edge/Edge interference	5
3.1.5	Edge/Face interference	6
3.1.6	Face/Face Interference	7
3.1.7	Computation Order	8
3.1.8	Results	8
3.2	Paves	8
3.3	Pave Blocks	9
3.4	Shrunk Range	9
3.5	Common Blocks	10
3.6	FacelInfo	10
<b>4</b>	<b>Data Structure</b>	<b>12</b>
4.1	Arguments	12
4.2	Shapes	12
4.3	Interferences	13
4.4	Pave, PaveBlock and CommonBlock	13
4.5	Points and Curves	14
4.6	FacelInfo	14
<b>5</b>	<b>Intersection Part</b>	<b>16</b>
5.1	Initialization	16
5.2	Compute Vertex/Vertex Interferences	16
5.3	Compute Vertex/Edge Interferences	17
5.4	Update Pave Blocks	18
5.5	Compute Edge/Edge Interferences	18
5.6	Compute Vertex/Face Interferences	20

5.7	Compute Edge/Face Interferences . . . . .	21
5.8	Build Split Edges . . . . .	22
5.9	Compute Face/Face Interferences . . . . .	22
5.10	Build Section Edges . . . . .	23
5.11	Build P-Curves . . . . .	25
5.12	Process Degenerated Edges . . . . .	25
<b>6</b>	<b>General description of the Building Part . . . . .</b>	<b>26</b>
<b>7</b>	<b>General Fuse Algorithm . . . . .</b>	<b>27</b>
7.1	Arguments . . . . .	27
7.2	Results . . . . .	27
7.3	Examples . . . . .	27
7.3.1	Example 1: Three edges intersecting at a point . . . . .	27
7.3.2	Example 2: Two wires and an edge . . . . .	28
7.3.3	Example 3: An edge intersecting with a face . . . . .	28
7.3.4	Example 4: An edge lying on a face . . . . .	29
7.3.5	Example 5: An edge and a shell . . . . .	29
7.3.6	Example 6: A wire and a shell . . . . .	30
7.3.7	Example 7: Three faces . . . . .	30
7.3.8	Example 8: A face and a shell . . . . .	30
7.3.9	Example 9: A shell and a solid . . . . .	31
7.3.10	Example 10: A compound and a solid . . . . .	32
7.4	Class BOPAlgo_Builder . . . . .	33
7.4.1	Fields . . . . .	33
7.4.2	Initialization . . . . .	33
7.4.3	Build Images for Vertices . . . . .	33
7.4.4	Build Result of Type Vertex . . . . .	34
7.4.5	Build Images for Edges . . . . .	34
7.4.6	Build Result of Type Edge . . . . .	34
7.4.7	Build Images for Wires . . . . .	34
7.4.8	Build Result of Type Wire . . . . .	35
7.4.9	Build Images for Faces . . . . .	35
7.4.10	Build Result of Type Face . . . . .	36
7.4.11	Build Images for Shells . . . . .	36
7.4.12	Build Result of Type Shell . . . . .	36
7.4.13	Build Images for Solids . . . . .	36
7.4.14	Build Result of Type Solid . . . . .	37
7.4.15	Build Images for Type CompSolid . . . . .	37
7.4.16	Build Result of Type Compsolid . . . . .	37
7.4.17	Build Images for Compounds . . . . .	37

7.4.18	Build Result of Type Compound	37
7.4.19	Post-Processing	37
<b>8</b>	<b>Boolean Operations Algorithm</b>	<b>39</b>
8.1	Arguments	39
8.2	Results	39
8.3	General Rules	39
8.4	Examples	40
8.4.1	Case 1: Two Vertices	40
8.4.2	Case 2: A Vertex and an Edge	40
8.4.3	Case 3: A Vertex and a Face	41
8.4.4	Case 4: A Vertex abd a Solid	42
8.4.5	Case 5: Two edges intersecting at one point	43
8.4.6	Case 6: Two edges having a common block	44
8.4.7	Case 7: An Edge and a Face intersecting at a point	45
8.4.8	Case 8: A Face and an Edge that have a common block	46
8.4.9	Case 9: An Edge and a Solid intersecting at a point	47
8.4.10	Case 10: An Edge and a Solid that have a common block	49
8.4.11	Case 11: Two intersecting faces	50
8.4.12	Case 12: Two faces that have a common part	52
8.4.13	Case 13: Two faces that have a common edge	53
8.4.14	Case 14: Two faces that have a common vertex	55
8.4.15	Case 15: A Face and a Solid that have an intersection curve.	56
8.4.16	Case 16: A Face and a Solid that have a common face.	57
8.4.17	Case 17: A Face and a Solid that have a common edge.	59
8.4.18	Case 18: A Face and a Solid that have a common vertex.	60
8.4.19	Case 19: Two intersecting Solids.	61
8.4.20	Case 20: Two Solids that have a shared face.	63
8.4.21	Case 21: Two Solids that have a shared edge.	64
8.4.22	Case 22: Two Solids that have a shared vertex.	66
8.5	Class BOPAlgo_BOP	68
8.6	Building Draft Result	69
8.7	Building the Result	69
<b>9</b>	<b>Algorithm Limitations</b>	<b>71</b>
9.1	Arguments	71
9.1.1	Common requirements	71
9.1.2	Pure self-interference	71
9.1.3	Self-interferences due to tolerances	73
9.1.4	Parametric representation	75
9.1.5	How to fix gaps using tolerance	77

9.2	Intersection problems . . . . .	78
9.2.1	Pure intersections and common zones . . . . .	78
9.2.2	Tolerances and inaccuracies . . . . .	79
9.2.3	Acquired Self-interferences . . . . .	81
<b>10</b>	<b>Appendix . . . . .</b>	<b>83</b>
10.1	Building faces from set of edges . . . . .	83
10.1.1	Terms and definitions . . . . .	83
10.1.2	Class BOPAlgo_BuilderArea . . . . .	83
10.1.3	Class BOPAlgo_BuilderFace . . . . .	84
10.1.4	Class BOPAlgo_WireSplitter . . . . .	84
10.2	Building solids from set of faces . . . . .	85
10.2.1	Terms and definitions . . . . .	85
10.2.2	Class BOPAlgo_BuilderSolid . . . . .	85
10.3	Packaging . . . . .	86

## 1 Introduction

This document provides comprehensive description of the Boolean Operation Algorithm (BOA) as implemented in Open CASCADE Technology. Boolean Operation Algorithm is based on General Fuse Algorithm (GFA) which is also described by this document. General Fuse Algorithm is also a basis of the Partition Algorithm (PA) implemented in SALOME and described in SALOME documentation.

General Fuse Algorithm has a history-based architecture designed to allow using OCAF naming functionality. The architecture of General Fuse Algorithm is expandable, that allows creating new algorithms basing on it.

## 2 Overview

### 2.1 Operators

#### 2.1.1 Boolean operator

The Boolean operator provides the operations (Common, Fuse, Cut) between two arguments in terms of *TopoDS\_Shape*.

The operator can be represented as:

$$R_B = B_j(S_1, S_2),$$

where:

- $B_j$  - operation of type  $j$  (Common, Fuse, Cut),
- $R_B$  - result of the operation,
- $S_1$  and  $S_2$  - arguments of the operation.

#### 2.1.2 General Fuse operator

The General fuse operator can be applied to an arbitrary number of arguments in terms of *TopoDS\_Shape*.

The GFA operator can be represented as:

$$R_{GF} = GF(S_1, S_2 \dots S_n),$$

where

- $R_{GF}$  - result of the operation,
- $S_1, S_2 \dots S_n$  - arguments of the operation,
- $n$  - number of arguments.

The result of the Boolean operator,  $R_B$ , can be obtained from  $R_{GF}$ .

For example, for two arguments  $S_1$  and  $S_2$  the result  $R_{GF}$  is

$$R_{GF} = GF(S_1, S_2) = S_{p1} + S_{p2} + S_{p12}$$

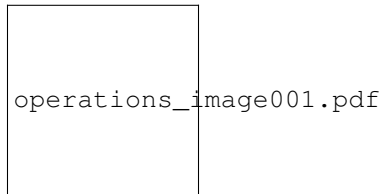


Figure 1: Operators

This Figure shows that

- $B_{common}(S_1, S_2) = S_{p12}$  ;
- $B_{cut12}(S_1, S_2) = S_{p1}$  ;
- $B_{cut21}(S_1, S_2) = S_{p2}$  ;
- $B_{fuse}(S_1, S_2) = S_{p1} + S_{p2} + S_{p12} = R_p$  ;

$$R_{GF} = GF(S_1, S_2) = B_{fuse} = B_{common} + B_{cut12} + B_{cut21}.$$

The fact that  $R_{GF}$  contains the components of  $R_B$  allows considering GFA as the general case of BOA. So it is possible to implement BOA as a subclass of GFA.

### 2.1.3 Partition operator

The Partition operator can be applied to an arbitrary number of arguments in terms of *TopoDS\_Shape*. The arguments are divided on two groups: Objects, Tools. The result of PA contains all parts belonging to the Objects but does not contain the parts that belongs to the Tools only.

The PA operator can be represented as follows:

$R_P = PA(S_{1O}, S_{2O} \dots S_{nO}, S_{1T}, S_{2T} \dots S_{nT})$ , where:

- $R_P$  - is the result of the operation;
- $S_{1O}, S_{2O} \dots S_{nO}$  - Arguments of the operation [Objects];
- $S_{1T}, S_{2T} \dots S_{nT}$  - Arguments of the operation [Tools];
- $nO$  - Number of arguments [Objects];
- $nT$  - Number of arguments [Tools].

The result  $R_P$  can be obtained from  $R_{GF}$ .

For example, for two arguments  $S_{1O}$  and  $S_{2T}$  the result  $R_P$  is

$$R_P = PA(S_{1O}, S_{2T}) = S_{p1} + S_{p12}.$$

In case when all arguments of the PA are Objects (no Tools), the result of PA is equivalent to the result of GFA.

For example, for two arguments  $S_{1O}$  and  $S_{2O}$  the result  $R_P$  is

$$R_P = PA(S_{1O}, S_{2O}) = S_{p1} + S_{p2} + S_{p12} = GF(S_1, S_2)$$

The fact that the  $R_{GF}$  contains the components of  $R_P$  allows considering GFA as the general case of PA. Thus, it is possible to implement PA as a subclass of GFA.

## 2.2 Parts of algorithms

GFA, BOA and PA have the same Data Structure (DS). The main goal of the Data Structure is to store all necessary information for input data and intermediate results.

GFA, BOA and PA consist of two main parts:

- Intersection Part (IP). The main goal of IP is to compute the interferences between sub-shapes of arguments. The IP uses DS to retrieve input data and store the results of intersections.
- Building Part (BP). The main goal of BP is to build required result of an operation. This part also uses DS to retrieve data and store the results.

As it follows from the definition of  $R_{GF}$  and  $R_P$  the main differences between GFA, BOA and PA are in the Building Part. The Intersection Part is the same for the algorithms.

## 3 Terms and Definitions

This chapter provides the background terms and definitions that are necessary to understand how the algorithms work.

### 3.1 Interferences

Each shape having a boundary representation (vertex, edge, face) contains an internal value of geometrical tolerance. The shapes interfere between each other in terms of their tolerances.

The shapes are interfered when there is a part of 3D space where the distance between the underlying geometry of shapes is less or equal to the sum of tolerances of the shapes. For the three types of shapes (vertex, edge, face) there are six types of interferences.

#### 3.1.1 Vertex/Vertex interference

Two vertices  $V_i$  and  $V_j$  have the distance between corresponding 3D points that is less than sum of the tolerances  $Tol(V_i)$  and  $Tol(V_j)$  of the vertices.

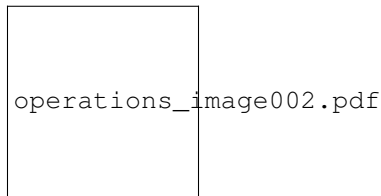


Figure 2: Vertex/vertex interference

The result is a new vertex  $V_n$  with 3D point  $P_n$  and tolerance value  $Tol(V_n)$  that are computed by the formulas:

$$P_n = 0.5 * (P_i + P_j)$$

$$Tol(V_n) = \max(Tol(V_i), Tol(V_j)) + 0.5 * D$$

where  $D = \text{distance}(P_i, P_j)$

#### 3.1.2 Vertex/Edge interference

For a vertex  $V_i$  and an edge  $E_j$  the distance  $D$  between 3D point of the vertex and its projection on the 3D curve of the edge  $E_j$  is less or equal than sum of tolerances of the vertex  $Tol(V_i)$  and the edge  $Tol(E_j)$ .

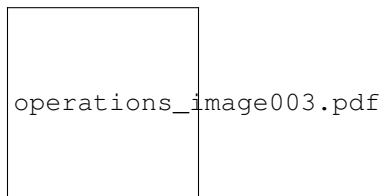


Figure 3: Vertex/edge interference

The result is vertex  $V_i$  with the corresponding tolerance value

$$Tol(V_i) = \max(Tol(V_i), Tol(E_j)) + 0.5 * D$$

where  $D = \text{distance}(Pi, PPi)$ ;

and parameter  $t_j$  of the projected point  $PPi$  on 3D curve  $Cj$  of edge  $Ej$ .

### 3.1.3 Vertex/Face interference

For a vertex  $Vi$  and a face  $Fj$  the distance  $D$  between 3D point of the vertex and its projection on the surface of the face is less or equal than sum of tolerances of the vertex  $Tol(Vi)$  and the face  $Tol(Fj)$ .

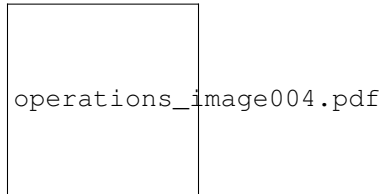


Figure 4: Vertex/face interference

The result is vertex  $Vi$  with the corresponding tolerance value:

$$Tol(Vi) = \max(Tol(Vi), Tol(Fj)) + 0.5 \cdot D$$

where  $D = \text{distance}(Pi, PPi)$

and parameters  $u_i, v_i$  of the projected point  $PPi$  on surface  $Sj$  of face  $Fj$ .

### 3.1.4 Edge/Edge interference

For two edges  $Ei$  and  $Ej$  (with the corresponding 3D curves  $Ci$  and  $Cj$ ) there are some places where the distance between the curves is less than (or equal to) sum of tolerances of the edges.

Let us examine two cases:

In the first case two edges have one or several common parts of 3D curves in terms of tolerance.



Figure 5: Edge/edge interference: common parts

The results are:

- Parametric range  $[t_{i1}, t_{i2}]$  for 3D curve  $Ci$  of edge  $Ei$ .
- Parametric range  $[t_{j1}, t_{j2}]$  for 3D curve  $Cj$  of edge  $Ej$ .

In the second case two edges have one or several common points in terms of tolerance.

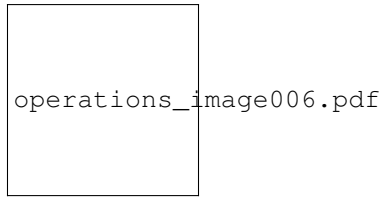


Figure 6: Edge/edge interference: common points

The results are as follows:

- one or several new vertices  $V_n$  with 3D point  $P_n$  and tolerance value  $Tol(V_n)$  that are calculated by the formulas:

$$P_n = 0.5 \cdot (P_i + P_j)$$

$$Tol(V_n) = \max(Tol(E_i), Tol(E_j)) + 0.5 \cdot D$$

where:

- $D = \text{distance}(P_i, P_j)$
- $P_i$  and  $P_j$  are the nearest 3D points for curves  $C_i$  and  $C_j$ .
- Parameter  $t_i$  of  $P_i$  for the 3D curve  $C_i$ .
- Parameter  $t_j$  of  $P_j$  for the 3D curve  $C_j$ .

### 3.1.5 Edge/Face interference

For an edge  $E_i$  (with the corresponding 3D curve  $C_i$ ) and a face  $F_j$  (with the corresponding 3D surface  $S_j$ ) there are some places in 3D space, where the distance between  $C_i$  and surface  $S_j$  is less than (or equal to) the sum of tolerances of edge  $E_i$  and face  $F_j$ .

Let us examine two cases:

In the first case Edge  $E_i$  and Face  $F_j$  have one or several common parts in terms of tolerance.



Figure 7: Edge/face interference: common parts

The result is a parametric range  $[t_{i1}, t_{i2}]$  for the 3D curve  $C_i$  of the edge  $E_i$ .

In the second case Edge  $E_i$  and Face  $F_j$  have one or several common points in terms of tolerance.



Figure 8: Edge/face interference: common points

The results are as follows:

- one or several new vertices  $V_n$  with 3D point  $P_n$  and tolerance value  $Tol(V_n)$  that are calculated by the formulas

$$P_n = 0.5 \cdot (P_i + P_j)$$

and

$$Tol(V_n) = \max(Tol(E_i), Tol(F_j)) + 0.5 \cdot D$$

where:

- $D = \text{distance}(P_i, P_j)$ ,
- $P_i$  and  $P_j$  are nearest 3D points for the curve  $C_i$  and surface  $S_j$ .
- Parameter  $t_i$  of  $P_i$  for the 3D curve  $C_i$ .
- Parameters  $u_j$  and  $v_j$  of the projected point  $PP_i$  on the surface  $S_j$  of the face  $F_j$ .

### 3.1.6 Face/Face Interference

For a face  $F_i$  and a face  $F_j$  (with the corresponding surfaces  $S_i$  and  $S_j$ ) there are some places in 3D space, where the distance between the surfaces is less than (or equal to) sum of tolerances of the faces.

In the first case Face  $F_i$  and face  $F_j$  have one or several common curves  $C_{ijk}$  ( $k = 0, 1, 2, \dots, k_N$ ) in terms of tolerance.

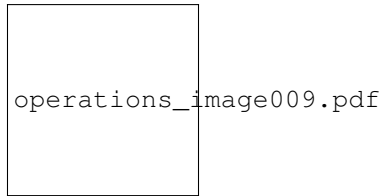


Figure 9: Face/face interference: common curves

The result is an intersection of curves  $C_{ijk}$  ( $k = 0, 1, 2, \dots, k_N$ , where  $k_N$  is the number of intersection curves) with corresponding values of tolerances  $Tol(C_{ijk})$ .

In the second case Face  $F_i$  and face  $F_j$  have one or several common points in terms of tolerance



Figure 10: Face/face interference: common points

The results are as follows:

- one or several new vertices  $V_{ijm}$  with 3D point  $P_n$  and tolerance value  $Tol(V_n)$  that are calculated by the formulas

$$P_n = 0.5 \cdot (P_i + P_j)$$

and

$$\text{Tol}(V_{ijm}) = \max(\text{Tol}(F_i), \text{Tol}(F_j)) + 0.5 \cdot D$$

where:

- $D = \text{distance}(P_i, P_j)$ ,
- $P_i$  and  $P_j$  are nearest 3D points for surfaces  $S_i$  and  $S_j$ .
- Parameters  $u_j, v_j$  of the projected point  $PP_j$  on surface  $S_j$  of face  $F_j$ ;
- Parameters  $u_i$  and  $v_i$  of the projected point  $PP_i$  on surface  $S_i$  of face  $F_i$ .

### 3.1.7 Computation Order

The interferences between shapes are computed on the basis of increasing of the dimension value of the shape in the following order:

- Vertex/Vertex,
- Vertex/Edge,
- Edge/Edge,
- Vertex/Face,
- Edge/Face,
- Face/Face.

This order allows avoiding the computation of redundant interferences between upper-level shapes  $S_i$  and  $S_j$  when there are interferences between lower sub-shapes  $S_{ik}$  and  $S_{jm}$ .

### 3.1.8 Results

- The result of the interference is a shape that can be either interfered shape itself (or its part) or a new shape.
- The result of the interference is a shape with the dimension value that is less or equal to the minimal dimension value of interfered shapes. For example, the result of Vertex/Edge interference is a vertex, but not an edge.
- The result of the interference splits the source shapes on the parts each time as it can do that.

## 3.2 Paves

The result of interferences of the type Vertex/Edge, Edge/Edge and Edge/Face in most cases is a vertex (new or old) lying on an edge.

The result of interferences of the type Face/Face in most cases is intersection curves, which go through some vertices lying on the faces.

The position of vertex  $V_i$  on curve  $C$  can be defined by a value of parameter  $t_i$  of the 3D point of the vertex on the curve. Pave  $PV_i$  on curve  $C$  is a structure containing the vertex  $V_i$  and correspondent value of the parameter  $t_i$  of the 3D point of the vertex on the curve. Curve  $C$  can be a 3D or a 2D curve.



Figure 11: Paves

Two paves  $PV1$  and  $PV2$  on the same curve  $C$  can be compared using the parameter value

$PV1 > PV2$  if  $t1 > t2$

The usage of paves allows binding of the vertex to the curve (or any structure that contains a curve: edge, intersection curve).

### 3.3 Pave Blocks

A set of paves  $PVi$  ( $i=1, 2...nPV$ ), where  $nPV$  is the number of paves] of curve  $C$  can be sorted in the increasing order using the value of parameter  $t$  on curve  $C$ .

A pave block  $PBi$  is a part of the object (edge, intersection curve) between neighboring paves.

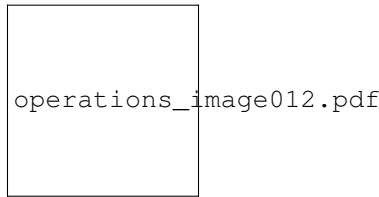


Figure 12: Pave Blocks

Any finite source edge  $E$  has at least one pave block that contains two paves  $PVb$  and  $PVe$ :

- Pave  $PVb$  corresponds to the vertex  $Vb$  with minimal parameter  $t_b$  on the curve of the edge.
- Pave  $PVe$  corresponds to the vertex  $Ve$  with maximal parameter  $t_e$  on the curve of the edge.

### 3.4 Shrunk Range

Pave block  $PV$  of curve  $C$  is bounded by vertices  $V1$  and  $V2$  with tolerance values  $Tol(V1)$  and  $Tol(V2)$ . Curve  $C$  has own value of tolerance  $Tol(C)$ :

- In case of edge, the value of this tolerance is tolerance of the edge.
- In case of intersection curve, the value of the tolerance is tolerance obtained from an intersection algorithm.

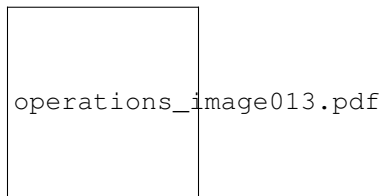


Figure 13: Shrunk Range

The theoretical parametric range of the pave block is  $[t1C, t2C]$ .

The positions of the vertices  $V1$  and  $V2$  of the pave block can be different. The positions are determined by the following conditions:

Distance (P1, P1c)  $Tol(V1) + Tol(C)$   
 Distance (P2, P2c)  $Tol(V2) + Tol(C)$

The Figure shows that each tolerance sphere of a vertex can reduce the parametric range of the pave block to a range  $[t1S, t2S]$ . The range  $[t1S, t2S]$  is the shrunk range of the pave block.

The shrunk range of the pave block is the part of 3D curve that can interfere with other shapes.

### 3.5 Common Blocks

The interferences of the type Edge/Edge, Edge/Face produce results as common parts.

In case of Edge/Edge interference the common parts are pave blocks that have different base edges.

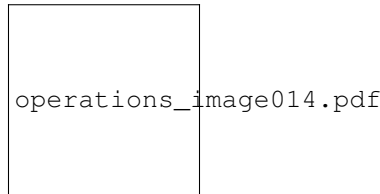


Figure 14: Common Blocks: Edge/Edge interference

If the pave blocks  $PB_1, PB_2 \dots PB_{NbPB}$ , where  $NbPB$  is the number of pave blocks have the same bounding vertices and geometrically coincide, the pave blocks form common block  $CB$ .

In case of Edge/Face interference the common parts are pave blocks lying on a face(s).



Figure 15: Common Blocks: Edge/Face interference

If the pave blocks  $PBi$  geometrically coincide with a face  $Fj$ , the pave blocks form common block  $CB$ .

In general case a common block  $CB$  contains:

- Pave blocks  $PBi$  ( $i=0, 1, 2, 3 \dots NbPB$ ).
- A set of faces  $Fj$  ( $j=0, 1 \dots NbF$ ),  $NbF$  - number of faces.

### 3.6 FacelInfo

The structure *FacelInfo* contains the following information:

- Pave blocks that have state **In** for the face;
- Vertices that have state **In** for the face;
- Pave blocks that have state **On** for the face;
- Vertices that have state **On** for the face;
- Pave blocks built up from intersection curves for the face;
- Vertices built up from intersection points for the face.

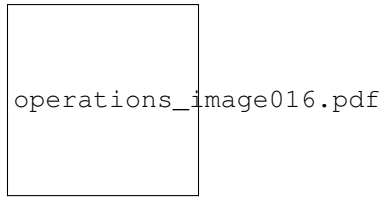


Figure 16: Face Info

In the figure, for face  $F1$ :

- Pave blocks that have state **In** for the face:  $PB_{in1}$  .
- Vertices that have state **In** for the face:  $V_{in1}$  .
- Pave blocks that have state **On** for the face:  $PB_{on11}$  ,  $PB_{on12}$  ,  $PB_{on2}$  ,  $PB_{on31}$  ,  $PB_{on32}$  ,  $PB_{on4}$  .
- Vertices that have state **On** for the face:  $V1$ ,  $V2$ ,  $V3$ ,  $V4$ ,  $V5$ ,  $V6$ .
- Pave blocks built up from intersection curves for the face:  $PB_{sc1}$  .
- Vertices built up from intersection points for the face: none

## 4 Data Structure

Data Structure (DS) is used to:

- Store information about input data and intermediate results;
- Provide the access to the information;
- Provide the links between the chunks of information.

This information includes:

- Arguments;
- Shapes (and sub-shapes);
- Interferences;
- Pave Blocks;
- Common Blocks.

Data Structure is implemented in the class *BOPDS\_DS*.

### 4.1 Arguments

The arguments are shapes (in terms of *TopoDS\_Shape*):

- Number of arguments is unlimited.
- Each argument is a valid shape (in terms of *BRepCheck\_Analyzer*).
- Each argument can be of one of the following types (see the Table):

No	Type	Index of Type
1	COMPOUND	0
2	COMPSOLID	1
3	SOLID	2
4	SHELL	3
5	FACE	4
6	WIRE	5
7	EDGE	6
8	VERTEX	7

- The argument of type 0 (*COMPOUND*) can include any number of shapes of an arbitrary type (0, 1...7).
- The argument should not be self-interfered, i.e. all sub-shapes of the argument that have geometrical coincidence through any topological entities (vertices, edges, faces) must share these entities.
- There are no restrictions on the type of underlying geometry of the shapes. The faces or edges of arguments  $S_i$  can have underlying geometry of any type supported by Open CASCADE Technology modeling algorithms (in terms of *GeomAbs\_CurveType* and *GeomAbs\_SurfaceType*).

### 4.2 Shapes

The information about Shapes is stored in structure *BOPDS\_ShapeInfo*. The objects of type *BOPDS\_ShapeInfo* are stored in the container of array type. The array allows getting the access to the information by an index (DS index). The structure *BOPDS\_ShapeInfo* has the following contents:

Name	Contents
<i>myShape</i>	Shape itself
<i>myType</i>	Type of shape
<i>myBox</i>	3D bounding box of the shape
<i>mySubShapes</i>	List of DS indices of sub-shapes
<i>myReference</i>	Storage for some auxiliary information
<i>myFlag</i>	Storage for some auxiliary information

### 4.3 Interferences

The information about interferences is stored in the instances of classes that are inherited from class *BOPDS\_Interf*.

Name	Contents
<i>BOPDS_Interf</i>	Root class for interference
<i>Index1</i>	DS index of the shape 1
<i>Index2</i>	DS index of the shape 2
<i>BOPDS_InterfVV</i>	Storage for Vertex/Vertex interference
<i>BOPDS_InterfVE</i>	Storage for Vertex/Edge interference
<i>myParam</i>	The value of parameter of the point of the vertex on the curve of the edge
<i>BOPDS_InterfVF</i>	Storage for Vertex/Face interference
<i>myU, myV</i>	The value of parameters of the point of the vertex on the surface of the face
<i>BOPDS_InterfEE</i>	Storage for Edge/Edge interference
<i>myCommonPart</i>	Common part (in terms of <i>IntTools_CommonPart</i> )
<i>BOPDS_InterfEF</i>	Storage for Edge/Face interference
<i>myCommonPart</i>	Common part (in terms of <i>*IntTools_CommonPart</i> )
<i>BOPDS_InterfFF</i>	Storage for Face/Face interference
<i>myTolR3D, myTolR2D</i>	The value of tolerances of curves (points) reached in 3D and 2D
<i>myCurves</i>	Intersection Curves (in terms of <i>BOPDS_Curve</i> )
<i>myPoints</i>	Intersection Points (in terms of <i>BOPDS_Point</i> )

The Figure shows inheritance diagram for *BOPDS\_Interf* classes.



Figure 17: BOPDS\_Interf classes

### 4.4 Pave, PaveBlock and CommonBlock

The information about the pave is stored in objects of type *BOPDS\_Pave*.

Name	Contents
<i>BOPDS_Pave</i>	
<i>myIndex1</i>	DS index of the vertex
<i>myParam</i>	Value of the parameter of the 3D point of vertex on curve.

The information about pave blocks is stored in objects of type *BOPDS\_PaveBlock*.

Name	Contents
<i>BOPDS_PaveBlock</i>	
<i>myEdge</i>	DS index of the edge produced from the pave block
<i>myOriginalEdge</i>	DS index of the source edge
<i>myPave1</i>	Pave 1 (in terms of <i>BOPDS_Pave</i> )
<i>myPave2</i>	Pave 2 (in terms of <i>BOPDS_Pave</i> )
<i>myExtPaves</i>	The list of paves (in terms of <i>BOPDS_Pave</i> ) that is used to store paves lying inside the pave block during intersection process
<i>myCommonBlock</i>	The reference to common block (in terms of <i>BOPDS_CommonBlock</i> ) if the pave block is a common block
<i>myShrunkData</i>	The shrunk range of the pave block

- To be bound to an edge (or intersection curve) the structures of type *BOPDS\_PaveBlock* are stored in one container of list type (*BOPDS\_ListOfPaveBlock*).
- In case of edge, all the lists of pave blocks above are stored in one container of array type. The array allows getting the access to the information by index of the list of pave blocks for the edge. This index (if exists) is stored in the field *myReference*.

The information about common block is stored in objects of type *BOPDS\_CommonBlock*.

Name	Contents
<i>BOPDS_CommonBlock</i>	
<i>myPaveBlocks</i>	The list of pave blocks that are common in terms of Common Blocks
<i>myFaces</i>	The list of DS indices of the faces, on which the pave blocks lie.

## 4.5 Points and Curves

The information about intersection point is stored in objects of type *BOPDS\_Point*.

Name	Contents
<i>BOPDS_Point</i>	
<i>myPnt</i>	3D point
<i>myPnt2D1</i>	2D point on the face1
<i>myPnt2D2</i>	2D point on the face2

The information about intersection curve is stored in objects of type *BOPDS\_Curve*.

Name	Contents
<i>BOPDS_Curve</i>	
<i>myCurve</i>	The intersection curve (in terms of <i>IntTools_Curve</i> )
<i>myPaveBlocks</i>	The list of pave blocks that belong to the curve
<i>myBox</i>	The bounding box of the curve (in terms of <i>Bnd_Box</i> )

## 4.6 FacelInfo

The information about *FacelInfo* is stored in a structure *BOPDS\_FacelInfo*. The structure *BOPDS\_FacelInfo* has the following contents.

Name	Contents
<i>BOPDS_FacelInfo</i>	
<i>myPaveBlocksIn</i>	Pave blocks that have state In for the face

<i>myVerticesIn</i>	Vertices that have state In for the face
<i>myPaveBlocksOn</i>	Pave blocks that have state On for the face
<i>myVerticesOn</i>	Vertices that have state On for the face
<i>myPaveBlocksSc</i>	Pave blocks built up from intersection curves for the face
<i>myVerticesSc</i>	Vertices built up from intersection points for the face +

The objects of type *BOPDS\_FacelInfo* are stored in one container of array type. The array allows getting the access to the information by index. This index (if exists) is stored in the field *myReference*.

## 5 Intersection Part

Intersection Part (IP) is used to

- Initialize the Data Structure;
- Compute interferences between the arguments (or their sub-shapes);
- Compute same domain vertices, edges;
- Build split edges;
- Build section edges;
- Build p-curves;
- Store all obtained information in DS. IP uses DS as input data. IP is implemented in the class *BOPAlgo\_PaveFiller*. The description provided in the next paragraphs is coherent with the implementation of the method *BOPAlgo\_PaveFiller::Perform()*.

### 5.1 Initialization

The input data for the step is the Arguments. The description of initialization step is shown in the Table.

No	Contents	Implementation
1	Initialization the array of shapes (in terms of Shapes). Filling the array of shapes.	<i>BOPDS_DS::Init()</i>
2	Initialization the array pave blocks (in terms of Pave, PaveBlock, CommonBlock)	<i>BOPDS_DS::Init()</i>
3	Initialization of intersection Iterator. The intersection Iterator is the object that computes intersections between BRep sub-shapes of the arguments in terms of bounding boxes. The intersection Iterator provides approximate number of the interferences for given type (in terms of Interferences)	<i>BOPDS_Iterator</i>
4	Initialization of intersection Context. The intersection Context is an object that contains geometrical and topological toolkit (classifiers, projectors, etc). The intersection Context is used to cache the tools to increase the algorithm performance.	<i>BOPInt_Context</i>

### 5.2 Compute Vertex/Vertex Interferences

The input data for this step is the DS after the Initialization. The description of this step is shown in the table :

No	Contents	Implementation
1	Initialize array of Vertex/Vertex interferences.	<i>BOPAlgo_PaveFiller::PerformVV()</i>

2	Access to the pairs of interfered shapes $(nVi, nVj)k, k=0, 1 \dots nk$ , where $nVi$ and $nVj$ are DS indices of vertices $Vi$ and $Vj$ and $nk$ is the number of pairs.	<i>BOPDS_Iterator</i>
3	Compute the connexity chains of interfered vertices $nV1C, nV2C \dots nVnC)k, C=0, 1 \dots nCs$ , where $nCs$ is the number of the connexity chains	<i>BOPAlgo_Tools::MakeBlocksCnx()</i>
4	Build new vertices from the chains $VNc, C=0, 1 \dots nCs$ .	<i>BOPAlgo_PaveFiller::PerformVV()</i>
5	Append new vertices in DS.	<i>BOPDS_DS::Append()</i>
6	Append same domain vertices in DS.	<i>BOPDS_DS::AddShapeSD()</i>
7	Append Vertex/Vertex interferences in DS.	<i>BOPDS_DS::AddInterf()</i>

- The pairs of interfered vertices are:  $(nV11, nV12), (nV11, nV13), (nV12, nV13), (nV13, nV15), (nV13, nV14), (nV14, nV15), (nV21, nV22), (nV21, nV23), (nV22, nV23)$ ;
- These pairs produce two chains:  $(nV11, nV12, nV13, nV14, nV15)$  and  $(nV21, nV22, nV23)$ ;
- Each chain is used to create a new vertex,  $VN1$  and  $VN2$ , correspondingly.

The example of connexity chains of interfered vertices is given in the image:



Figure 18: Connexity chains of interfered vertices

### 5.3 Compute Vertex/Edge Interferences

The input data for this step is the DS after computing Vertex/Vertex interferences.

No	Contents	Implementation
1	Initialize array of Vertex/Edge interferences	<i>BOPAlgo_PaveFiller::PerformVE()</i>
2	Access to the pairs of interfered shapes $(nVi, nEj)k k=0, 1 \dots nk$ , where $nVi$ is DS index of vertex $Vi$ , $nEj$ is DS index of edge $Ej$ and $nk$ is the number of pairs.	<i>BOPDS_Iterator</i>
3	Compute paves. See Vertex/Edge Interference	<i>BOPInt_Context::ComputeVE()</i>
4	Initialize pave blocks for the edges $Ej$ involved in the interference	<i>BOPDS_DS::ChangePaveBlocks()</i>

5	Append the paves into the pave blocks in terms of Pave, PaveBlock and CommonBlock	<i>BOPDS_PaveBlock::AppendExtPave()</i>
6	Append Vertex/Edge interferences in DS	<i>BOPDS_DS::AddInterf()</i>

#### 5.4 Update Pave Blocks

The input data for this step is the DS after computing Vertex/Edge Interferences.

No	Contents	Implementation
1	Each pave block PB containing internal paves is split by internal paves into new pave blocks <i>PBN1</i> , <i>PBN2</i> ... <i>PBNn</i> . PB is replaced by new pave blocks <i>PBN1</i> , <i>PBN2</i> ... <i>PBNn</i> in the DS.	<i>BOPDS_DS::UpdatePaveBlocks()</i>

#### 5.5 Compute Edge/Edge Interferences

The input data for this step is the DS after updating Pave Blocks.

No	Contents	Implementation
1	Initialize array of Edge/Edge interferences	<i>BOPAlgo_PaveFiller::PerformEE()</i>
2	Access to the pairs of interfered shapes $(nEi, nEj)k, k=0, 1 \dots nk$ , where $nEi$ is DS index of the edge $Ei$ , $nEj$ is DS index of the edge $Ej$ and $nk$ is the number of pairs.	<i>BOPDS_Iterator</i>
3	Initialize pave blocks for the edges involved in the interference, if it is necessary.	<i>BOPDS_DS::ChangePaveBlocks()</i>
4	Access to the pave blocks of interfered shapes: $(PBi1, PBi2 \dots PBiNi)$ for edge $Ei$ and $(PBj1, PBj2 \dots PBjNj)$ for edge $Ej$	<i>BOPAlgo_PaveFiller::PerformEE()</i>
5	Compute shrunk data for pave blocks in terms of Pave, PaveBlock and CommonBlock, if it is necessary.	<i>BOPAlgo_PaveFiller::FillShrunk-Data()</i>
6	Compute Edge/Edge interference for pave blocks $PBix$ and $PBiy$ . The result of the computation is a set of objects of type <i>IntTools_CommonPart</i>	<i>IntTools_EdgeEdge</i>
7.1	For each <i>CommonPart</i> of type <i>VERTEX</i> : Create new vertices $VNi$ ( $i=1, 2 \dots NbVN$ ), where $NbVN$ is the number of new vertices. Intersect the vertices $VNi$ using the steps Initialization and compute Vertex/Vertex interferences as follows: a) create a new object <i>PFn</i> of type <i>BOPAlgo_PaveFiller</i> with its own DS; b) use new vertices $VNi$ ( $i=1, 2 \dots NbVN$ ), $NbVN$ as arguments (in terms of <i>TopoDs_Shape</i> ) of <i>PFn</i> ; c) invoke method <i>Perform()</i> for <i>PFn</i> . The resulting vertices $VNXi$ ( $i=1, 2 \dots NbVNX$ ), where $NbVNX$ is the number of vertices, are obtained via mapping between $VNi$ and the results of <i>PVn</i> .	<i>BOPTools_Tools::MakeNew-Vertex()</i>
7.2	For each <i>CommonPart</i> of type <i>EDGE</i> : Compute the coinciding connexity chains of pave blocks $(PB1C, PB2C \dots PNnC)k, C=0, 1 \dots nCs$ , where $nCs$ is the number of the connexity chains. Create common blocks $(CBc, C=0, 1 \dots nCs)$ from the chains. Attach the common blocks to the pave blocks.	<i>BOPAlgo_Tools::PerformCommon-Blocks()</i>

8	Post-processing. Append the paves of $VNX_i$ into the corresponding pave blocks in terms of Pave, PaveBlock and CommonBlock	<i>BOPDS_PaveBlock::AppendExtPave()</i>
9	Split common blocks CBc by the paves.	<i>BOPDS_DS::UpdateCommonBlock()</i>
10	Append Edge/Edge interferences in the DS.	<i>BOPDS_DS::AddInterf()</i>

The example of coinciding chains of pave blocks is given in the image:

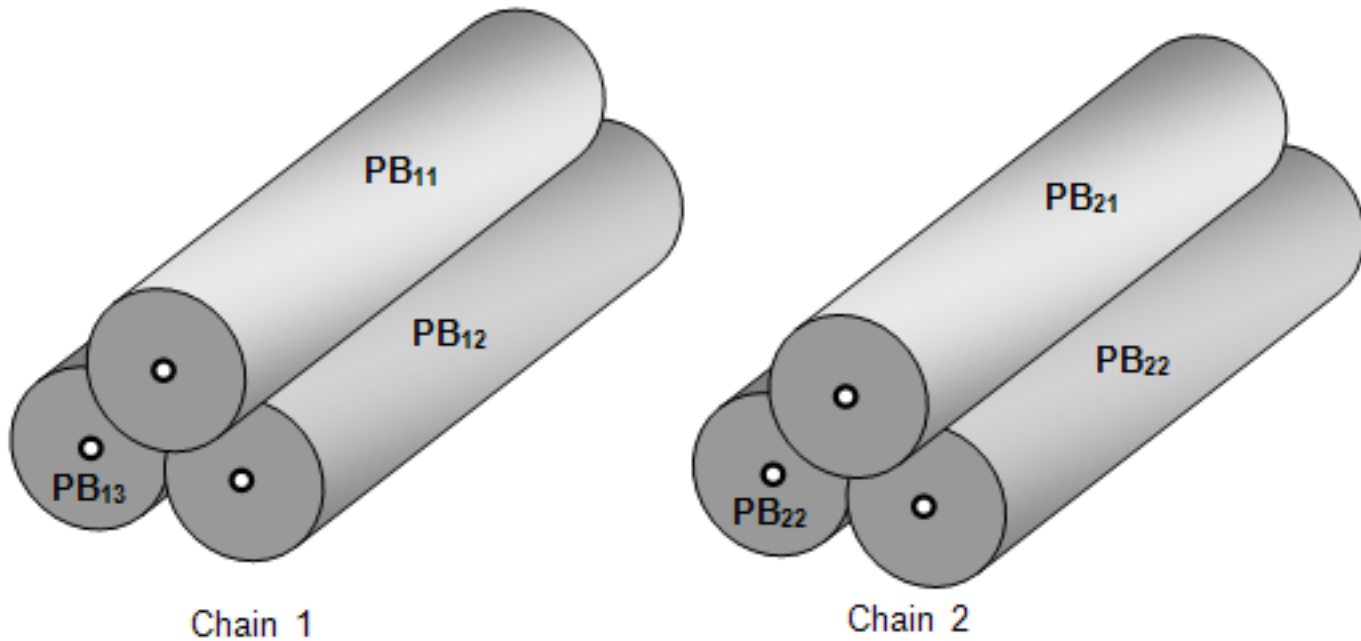


Figure 19: Coinciding chains of pave blocks

- The pairs of coincided pave blocks are:  $(PB_{11}, PB_{12})$ ,  $(PB_{11}, PB_{13})$ ,  $(PB_{12}, PB_{13})$ ,  $(PB_{21}, PB_{22})$ ,  $(PB_{21}, PB_{23})$ ,  $(PB_{22}, PB_{23})$ .
- The pairs produce two chains:  $(PB_{11}, PB_{12}, PB_{13})$  and  $(PB_{21}, PB_{22}, PB_{23})$ .

## 5.6 Compute Vertex/Face Interferences

The input data for this step is the DS after computing Edge/Edge interferences.

No	Contents	Implementation
1	Initialize array of Vertex/Face interferences	<i>BOPAlgo_PaveFiller::PerformVF()</i>
2	Access to the pairs of interfered shapes $(nVi, nFj)k$ , $k=0, 1 \dots nk$ , where $nVi$ is DS index of the vertex $Vi$ , $nFj$ is DS index of the edge $Fj$ and $nk$ is the number of pairs.	<i>BOPDS_Iterator</i>

3	Compute interference See Vertex/Face Interference	<i>BOPInt_Context::ComputeVF()</i>
4	Append Vertex/Edge interferences in the DS	<i>BOPDS_DS::AddInterf()</i>
5	Repeat steps 2-4 for each new vertex $VNX_i$ ( $i=1, 2, \dots, NbVNX$ ), where $NbVNX$ is the number of vertices.	<i>BOPAlgo_PaveFiller::Treat-VerticesEE()</i>

## 5.7 Compute Edge/Face Interferences

The input data for this step is the DS after computing Vertex/Face Interferences.

No	Contents	Implementation
1	Initialize array of Edge/Face interferences	<i>BOPAlgo_PaveFiller::PerformEF()</i>
2	Access to the pairs of interfered shapes $(nEi, nFj)k$ , $k=0, 1, \dots, nk$ , where $nEi$ is DS index of edge $Ei$ , $nFj$ is DS index of face $Fj$ and $nk$ is the number of pairs.	<i>BOPDS_Iterator</i>
3	Initialize pave blocks for the edges involved in the interference, if it is necessary.	<i>BOPDS_DS::ChangePaveBlocks()</i>
4	Access to the pave blocks of interfered edge $(PBi1, PBi2, \dots, PBiNi)$ for edge $Ei$	<i>BOPAlgo_PaveFiller::PerformEF()</i>
5	Compute shrunk data for pave blocks (in terms of Pave, PaveBlock and CommonBlock) if it is necessary.	<i>BOPAlgo_PaveFiller::FillShrunk-Data()</i>
6	Compute Edge/Face interference for pave block $PBix$ , and face $nFj$ . The result of the computation is a set of objects of type <i>IntTools_CommonPart</i>	<i>IntTools_EdgeFace</i>
7.1	For each <i>CommonPart</i> of type VERTEX: Create new vertices $VNi$ ( $i=1, 2, \dots, NbVN$ ), where $NbVN$ is the number of new vertices. Merge vertices $VNi$ as follows: a) create new object $PFn$ of type <i>BOPAlgo_PaveFiller</i> with its own DS; b) use new vertices $VNi$ ( $i=1, 2, \dots, NbVN$ ), $NbVN$ as arguments (in terms of <i>TopoDs_Shape</i> ) of $PFn$ ; c) invoke method <i>Perform()</i> for $PFn$ . The resulting vertices $VNX_i$ ( $i=1, 2, \dots, NbVNX$ ), where $NbVNX$ is the number of vertices, are obtained via mapping between $VNi$ and the results of $PVn$ .	<i>BOPTools_Tools::MakeNew-Vertex()</i> and <i>BOPAlgo_PaveFiller::Perform-Vertices1()</i>

7.2	For each <i>CommonPart</i> of type <i>EDGE</i> : Create common blocks ( <i>CBc</i> . $C=0, 1 \dots nCs$ ) from pave blocks that lie on the faces. Attach the common blocks to the pave blocks.	<i>BOPAlgo_Tools::PerformCommon-Blocks()</i>
8	Post-processing. Append the paves of <i>VNXi</i> into the corresponding pave blocks in terms of <i>Pave</i> , <i>PaveBlock</i> and <i>CommonBlock</i> .	<i>BOPDS_PaveBlock::AppendExtPave()</i>
9	Split pave blocks and common blocks <i>CBc</i> by the paves.	<i>BOPAlgo_PaveFiller::Perform-Vertices1()</i> , <i>BOPDS_DS::UpdatePaveBlock()</i> and <i>BOPDS_DS::UpdateCommonBlock()</i>
10	Append Edge/Face interferences in the DS	<i>BOPDS_DS::AddInterf()</i>
11	Update <i>FaceInfo</i> for all faces having EF common parts.	<i>BOPDS_DS::UpdateFaceInfoIn()</i>

## 5.8 Build Split Edges

The input data for this step is the DS after computing Edge/Face Interferences.

For each pave block *PB* take the following steps:

No	Contents	Implementation
1	Get the real pave block <i>PBR</i> , which is equal to <i>PB</i> if <i>PB</i> is not a common block and to <i>PB<sub>1</sub></i> if <i>PB</i> is a common block. <i>PB<sub>1</sub></i> is the first pave block in the pave blocks list of the common block. See <i>Pave</i> , <i>PaveBlock</i> and <i>CommonBlock</i> .	<i>BOPAlgo_PaveFiller::MakeSplit-Edges()</i>
2	Build the split edge <i>Esp</i> using the information from <i>DS</i> and <i>PBR</i> .	<i>BOPTools_Tools::MakeSplitEdge()</i>
3	Compute <i>BOPDS_ShapeInfo</i> contents for <i>Esp</i>	<i>BOPAlgo_PaveFiller::MakeSplit-Edges()</i>
4	Append <i>BOPDS_ShapeInfo</i> contents to the DS	<i>BOPDS_DS::Append()</i>

## 5.9 Compute Face/Face Interferences

The input data for this step is DS after building Split Edges.

No	Contents	Implementation
1	Initialize array of Edge/Face interferences	<i>BOPAlgo_PaveFiller::PerformFF()</i>
2	Access to the pairs of interfered shapes ( <i>nFi</i> , <i>nFj</i> ) $k, k=0, 1 \dots nk$ , where <i>nFi</i> is DS index of edge <i>Fi</i> , <i>nFj</i> is DS index of face <i>Fj</i> and <i>nk</i> is the number of pairs.	<i>BOPDS_Iterator</i>

3	Compute Face/Face interference	<i>IntTools_FaceFace</i>
4	Append Face/Face interferences in the DS.	<i>BOPDS_DS::AddInterf()</i>

### 5.10 Build Section Edges

The input data for this step is the DS after building Split Edges.

No	Contents	Implementation
1	For each Face/Face interference $nFi$ , $nFj$ , retrieve <i>FaceInfo</i> . Create draft vertices from intersection points $VPk$ ( $k=1, 2, \dots, NbVP$ ), where $NbVP$ is the number of new vertices, and the draft vertex $VPk$ is created from an intersection point if $VPk \neq Vm$ ( $m = 0, 1, 2, \dots, NbVm$ ), where $Vm$ is an existing vertex for the faces $nFi$ and $nFj$ ( <i>On</i> or <i>In</i> in terms of <i>TopoDs_Shape</i> ), $NbVm$ is the number of vertices existing on faces $nFi$ and $nFj$ and $\neq$ means non-coincidence in terms of Vertex/Vertex interference.	<i>BOPAlgo_PaveFiller::MakeBlocks()</i>
2	For each intersection curve $Cijk$	
2.1	Create paves $PVc$ for the curve using existing vertices, i.e. vertices <i>On</i> or <i>In</i> (in terms of <i>FaceInfo</i> ) for faces $nFi$ and $nFj$ . Append the paves $PVc$	<i>BOPAlgo_PaveFiller::PutPaveOnCurve()</i> and <i>BOPDS_PaveBlock::AppendExtPave()</i>
2.2	Create technological vertices $Vt$ , which are the bounding points of an intersection curve (with the value of tolerance $Tol(Cijk)$ ). Each vertex $Vt$ with parameter $Tt$ on curve $Cijk$ forms pave $PVt$ on curve $Cijk$ . Append technological paves.	<i>BOPAlgo_PaveFiller::PutBoundPaveOnCurve()</i>
2.3	Create pave blocks $PBk$ for the curve using paves ( $k=1, 2, \dots, NbPB$ ), where $NbPB$ is the number of pave blocks	<i>BOPAlgo_PaveFiller::MakeBlocks()</i>
2.4	Build draft section edges $ESk$ using the pave blocks ( $k=1, 2, \dots, NbES$ ), where $NbES$ is the number of draft section edges. The draft section edge is created from a pave block $PBk$ if $PBk$ has state <i>In</i> or <i>On</i> for both faces $nFi$ and $nFj$ and $PBk \neq PBm$ ( $m=0, 1, 2, \dots, NbPBm$ ), where $PBm$ is an existing pave block for faces $nFi$ and $nFj$ ( <i>On</i> or <i>In</i> in terms of <i>FaceInfo</i> ), $NbVm$ is the number of existing pave blocks for faces $nFi$ and $nFj$ and $\neq$ means non-coincidence (in terms of Vertex/Face interference).	<i>BOPTools_Tools::MakeEdge()</i>
3	Intersect the draft vertices $VPk$ ( $k=1, 2, \dots, NbVP$ ) and the draft section edges $ESk$ ( $k=1, 2, \dots, NbES$ ). For this: a) create new object $PFn$ of type <i>BOPAlgo_PaveFiller</i> with its own DS; b) use vertices $VPk$ and edges $ESk$ as arguments (in terms of Arguments) of $PFn$ ; c) invoke method <i>Perform()</i> for $PFn$ . Resulting vertices $VPXk$ ( $k=1, 2, \dots, NbVPX$ ) and edges $ESXk$ ( $k=1, 2, \dots, NbESX$ ) are obtained via mapping between $VPk$ , $ESk$ and the results of $PFn$ .	<i>BOPAlgo_PaveFiller::PostTreatF()</i>
(c) Open CASCADE 2014		

4	Update face info (sections about pave blocks and vertices)	<i>BOPAlgo_PaveFiller::PerformFF()</i>
---	--	--

### 5.11 Build P-Curves

The input data for this step is the DS after building section edges.

No	Contents	Implementation
1	For each Face/Face interference $nFi$ and $nFj$ build p-Curves on $nFi$ and $nFj$ for each section edge $ESXk$ .	<i>BOPAlgo_PaveFiller::MakeP-Curves()</i>
2	For each pave block that is common for faces $nFi$ and $nFj$ build p-Curves on $nFi$ and $nFj$ .	<i>BOPAlgo_PaveFiller::MakeP-Curves()</i>

### 5.12 Process Degenerated Edges

The input data for this step is the DS after building P-curves.

No	Contents	Implementation
	For each degenerated edge $ED$ having vertex $VD$	<i>BOPAlgo_PaveFiller::ProcessDE()</i>
1	Find pave blocks $PBi$ ( $i=1,2,\dots,NbPB$ ), where $NbPB$ is the number of pave blocks, that go through vertex $VD$ .	<i>BOPAlgo_PaveFiller::FindPave-Blocks()</i>
2	Compute paves for the degenerated edge $ED$ using a 2D curve of $ED$ and a 2D curve of $PBi$ . Form pave blocks $PBDi$ ( $i=1,2,\dots,NbPBD$ ), where $NbPBD$ is the number of the pave blocks for the degenerated edge $ED$	<i>BOPAlgo_PaveFiller::FillPaves()</i>
3	Build split edges $ESDi$ ( $i=1,2,\dots,NbESD$ ), where $ESD$ is the number of split edges, using the pave blocks $PBDi$	<i>BOPAlgo_PaveFiller::MakeSplitEdge()</i>

## 6 General description of the Building Part

Building Part (BP) is used to

- Build the result of the operation
- Provide history information (in terms of *::Generated()*, *::Modified()* and *::IsDeleted()*) BP uses the DS prepared by *BOPAlgo\_PaveFiller* described at chapter 5 as input data. BP is implemented in the following classes:
- *BOPAlgo\_Builder* - for the General Fuse Algorithm (GFA).
- *BOPAlgo\_BOP* - for the Boolean Operation Algorithm (BOA).



Figure 20: Diagram for BP classes

The class *BOPAlgo\_Algo* provides the base interface for all algorithms:

- Error status;
- Warning status;
- Checking data;
- Checking the result;
- Memory allocator. The class *BOPAlgo\_BuilderShape* provides the interface for algorithms that have:
- A Shape as the result;
- History information (in terms of *::Generated()*, *::Modified()* and *::IsDeleted()*).

## 7 General Fuse Algorithm

### 7.1 Arguments

The arguments of the algorithm are shapes (in terms of *TopoDS\_Shape*). The main requirements for the arguments are described in Algorithms chapter.

### 7.2 Results

During the operation argument  $S_i$  can be split into several parts  $S_{i1}, S_{i2} \dots S_{iNbSp}$ , where  $NbSp$  is the number of parts. The set  $(S_{i1}, S_{i2} \dots S_{iNbSp})$  is an image of argument  $S_i$ .

- The result of the General Fuse (GF) operation is a compound. Each sub-shape of the compound (resulting shape) corresponds to the certain argument shape  $S_1, S_2 \dots S_n$  and has shared sub-shapes in accordance with interferences between the arguments.
- For the arguments of the type EDGE, FACE, SOLID the result contains split parts of the argument.
- For the arguments of the type WIRE, SHELL, COMPSOLID, COMPOUND the result contains the image of the shape of the corresponding type (i.e. WIRE, SHELL, COMPSOLID or COMPOUND). The types of resulting shapes depend on the type of the corresponding argument participating in the operation. See the table below:

No	Type of argument	Type of resulting shape	Comments
1	COMPOUND	COMPOUND	The resulting COMPOUND is built from images of sub-shapes of type COMPOUND COMPSOLID, SHELL, WIRE and VERTEX. Sets of split sub-shapes of type SOLID, FACE, EDGE.
2	COMPSOLID	COMPSOLID	The resulting COMPSOLID is built from split SOLIDs.
3	SOLID	Set of split SOLIDs	
4	SHELL	SHELL	The resulting SHELL is built from split FACES
5	FACE	Set of split FACES	
6	WIRE	WIRE	The resulting WIRE is built from split EDGES
7	EDGE	Set of split EDGES	
8	VERTEX	VERTEX	

### 7.3 Examples

Please, have a look at the examples of compounds, which can help to better understand the definitions.

#### 7.3.1 Example 1: Three edges intersecting at a point

Let us consider three edges:  $E_1$ ,  $E_2$  and  $E_3$  that intersect in one 3D point.



Figure 21: Three Intersecting Edges

The result of the GFA operation is a compound containing 6 new edges:  $E11$ ,  $E12$ ,  $E21$ ,  $E22$ ,  $E31$ , and  $E32$ . These edges have one shared vertex  $Vn1$ .

In this case:

- The argument edge  $E1$  has resulting split edges  $E11$  and  $E12$  (image of  $E1$ ).
- The argument edge  $E2$  has resulting split edges  $E21$  and  $E22$  (image of  $E2$ ).
- The argument edge  $E3$  has resulting split edges  $E31$  and  $E32$  (image of  $E3$ ).

### 7.3.2 Example 2: Two wires and an edge

Let us consider two wires  $W1$  ( $Ew11$ ,  $Ew12$ ,  $Ew13$ ) and  $W2$  ( $Ew21$ ,  $Ew22$ ,  $Ew23$ ) and edge  $E1$ .



Figure 22: Two wires and an edge

The result of the GF operation is a compound consisting of 2 wires:  $Wn1$  ( $Ew11$ ,  $En1$ ,  $En2$ ,  $En3$ ,  $Ew13$ ) and  $Wn2$  ( $Ew21$ ,  $En2$ ,  $En3$ ,  $En4$ ,  $Ew23$ ) and two edges:  $E11$  and  $E12$ .

In this case :

- The argument  $W1$  has image  $Wn1$ .
- The argument  $W2$  has image  $Wn2$ .
- The argument edge  $E1$  has split edges  $E11$  and  $E12$ . (image of  $E1$ ). The edges  $En1$ ,  $En2$ ,  $En3$ ,  $En4$  and vertex  $Vn1$  are new shapes created during the operation. Edge  $Ew12$  has split edges  $En1$ ,  $En2$  and  $En3$  and edge  $Ew22$  has split edges  $En2$ ,  $En3$  and  $En4$ .

### 7.3.3 Example 3: An edge intersecting with a face

Let us consider edge  $E1$  and face  $F2$ :

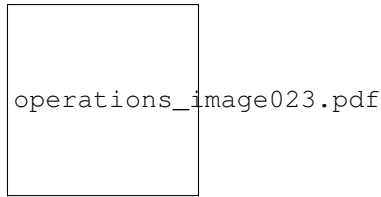


Figure 23: An edge intersecting with a face

The result of the GF operation is a compound consisting of 3 shapes:

- Split edge parts  $E11$  and  $E12$  (image of  $E1$ ).
- New face  $F21$  with internal edge  $E12$  (image of  $F2$ ).

#### 7.3.4 Example 4: An edge lying on a face

Let us consider edge  $E1$  and face  $F2$ :



Figure 24: An edge lying on a face

The result of the GF operation is a compound consisting of 5 shapes:

- Split edge parts  $E11$ ,  $E12$  and  $E13$  (image of  $E1$ ).
- Split face parts  $F21$  and  $F22$  (image of  $F2$ ).

#### 7.3.5 Example 5: An edge and a shell

Let us consider edge  $E1$  and shell  $Sh2$  that consists of 2 faces:  $F21$  and  $F22$



Figure 25: An edge and a shell

The result of the GF operation is a compound consisting of 5 shapes:

- Split edge parts  $E11$ ,  $E12$ ,  $E13$  and  $E14$  (image of  $E1$ ).
- Image shell  $Sh21$  (that contains split face parts  $F211$ ,  $F212$ ,  $F221$  and  $F222$ ).

## 7.3.6 Example 6: A wire and a shell

Let us consider wire  $W1$  ( $E1, E2, E3, E4$ ) and shell  $Sh2$  ( $F21, F22$ ).



Figure 26: A wire and a shell

The result of the GF operation is a compound consisting of 2 shapes:

- Image wire  $W11$  that consists of split edge parts from wire  $W1$ :  $E11, E12, E13$  and  $E14$ .
- Image shell  $Sh21$  that contains split face parts:  $F211, F212, F213, F221, F222$  and  $F223$ .

## 7.3.7 Example 7: Three faces

Let us consider 3 faces:  $F1, F2$  and  $F3$ .

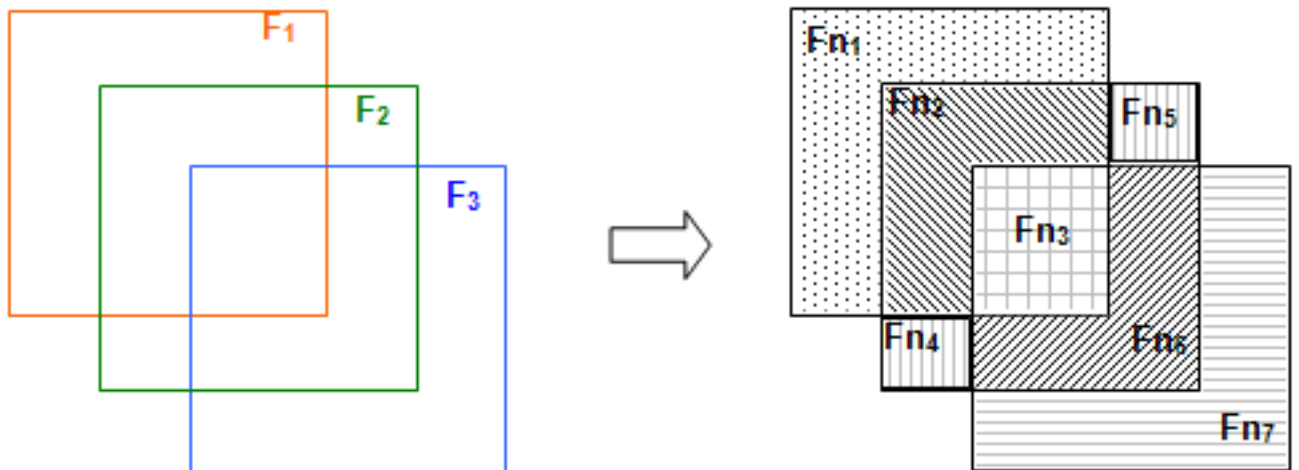


Figure 27: Three faces

The result of the GF operation is a compound consisting of 7 shapes:

- Split face parts:  $Fn1, Fn2, Fn3, Fn4, Fn5, Fn6$  and  $Fn7$ .

## 7.3.8 Example 8: A face and a shell

Let us consider shell  $Sh1$  ( $F11, F12, F13$ ) and face  $F2$ .

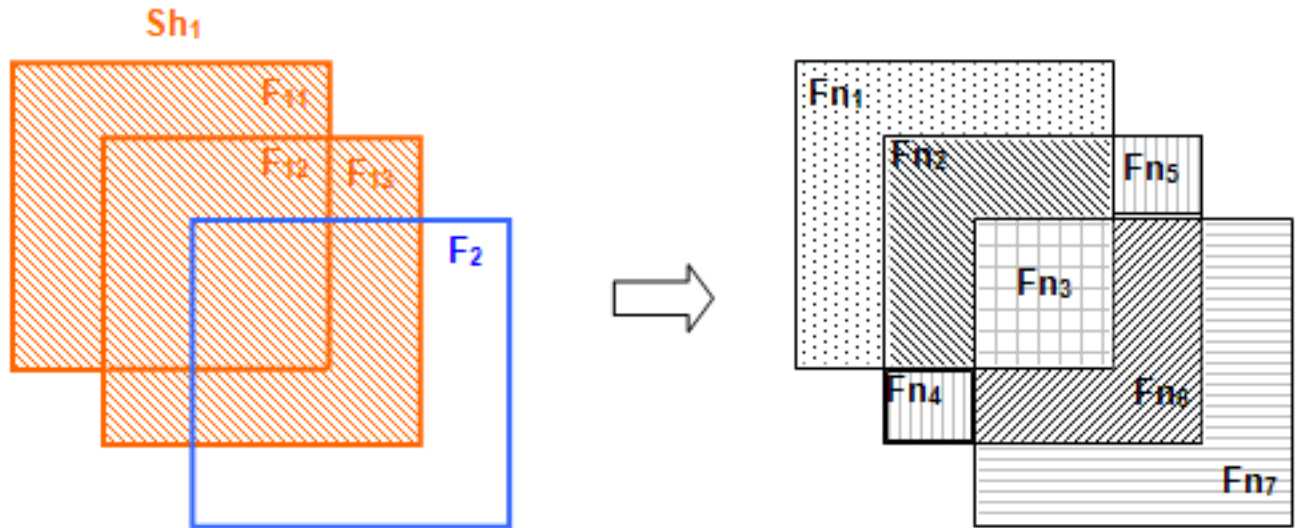


Figure 28: A face and a shell

The result of the GF operation is a compound consisting of 4 shapes:

- Image shell  $Sh11$  that consists of split face parts from shell  $Sh1$ :  $Fn1$ ,  $Fn2$ ,  $Fn3$ ,  $Fn4$ ,  $Fn5$  and  $Fn6$ .
- Split parts of face  $F2$ :  $Fn3$ ,  $Fn6$  and  $Fn7$ .

#### 7.3.9 Example 9: A shell and a solid

Let us consider shell  $Sh1$  ( $F11$ ,  $F12 \dots F16$ ) and solid  $So2$ .

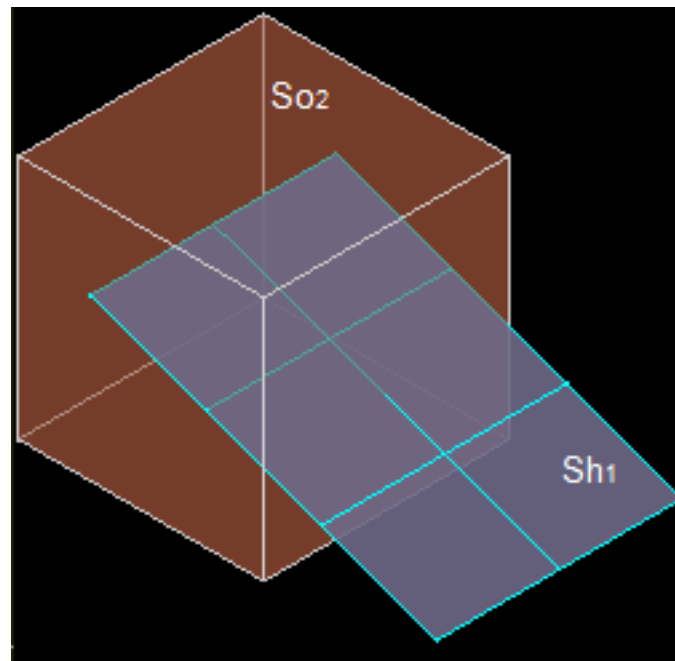


Figure 29: A shell and a solid: arguments

The result of the GF operation is a compound consisting of 2 shapes:

- Image shell  $Sh11$  consisting of split face parts of  $Sh1$ :  $Fn1, Fn2 \dots Fn8$ .
- Solid  $So21$  with internal shell. (image of  $So2$ ).

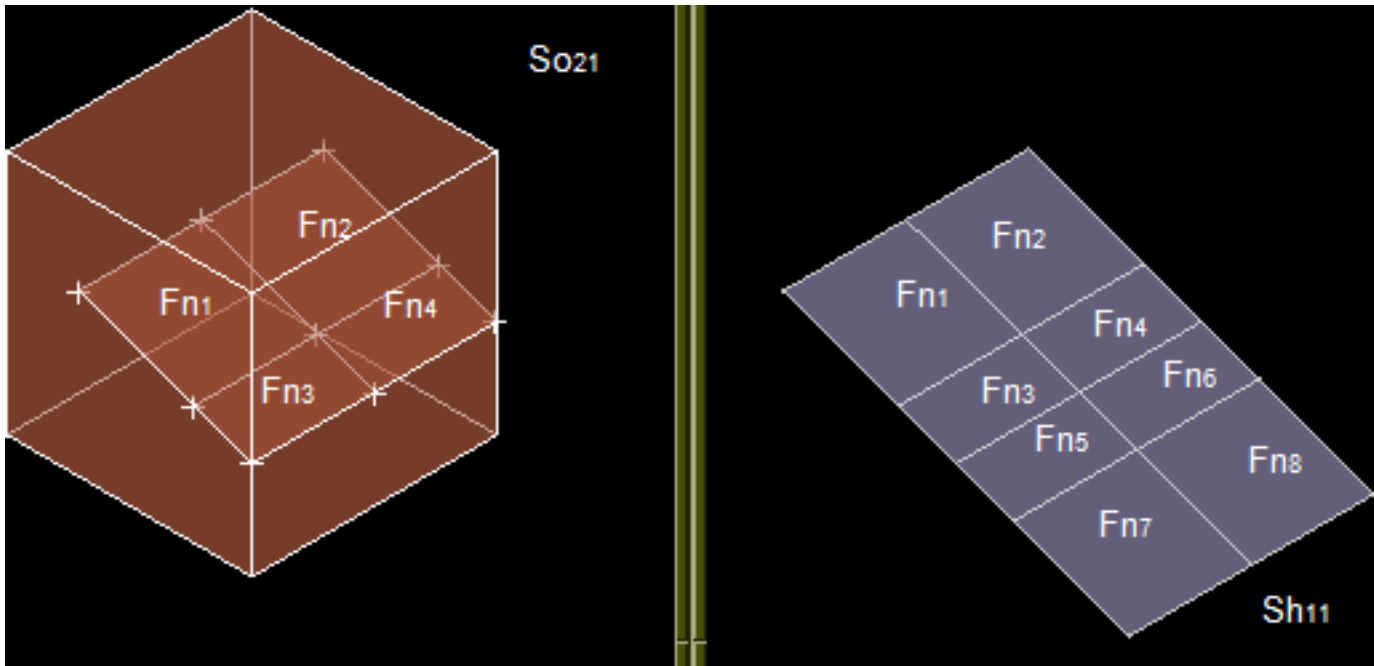


Figure 30: A shell and a solid: results

#### 7.3.10 Example 10: A compound and a solid

Let us consider compound  $Cm1$  consisting of 2 solids  $So11$  and  $So12$ ) and solid  $So2$ .

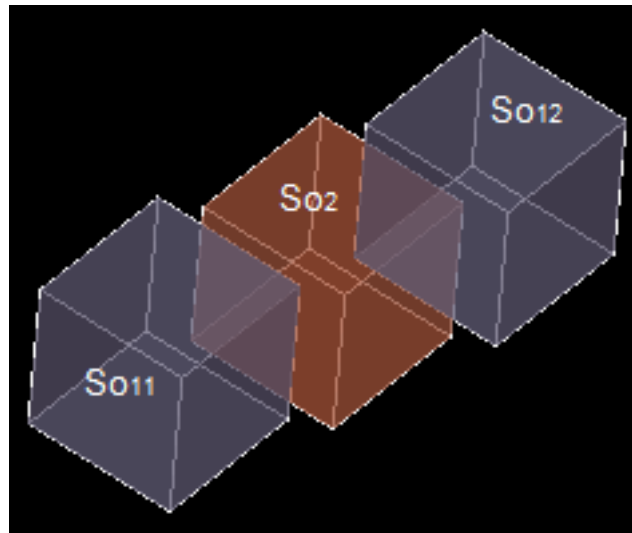


Figure 31: A compound and a solid: arguments

The result of the GF operation is a compound consisting of 4 shapes:

- Image compound  $Cm11$  consisting of split solid parts from  $So11$  and  $So12$  ( $Sn1, Sn2, Sn3, Sn4$ ).
- Split parts of solid  $So2$  ( $Sn2, Sn3, Sn5$ ).

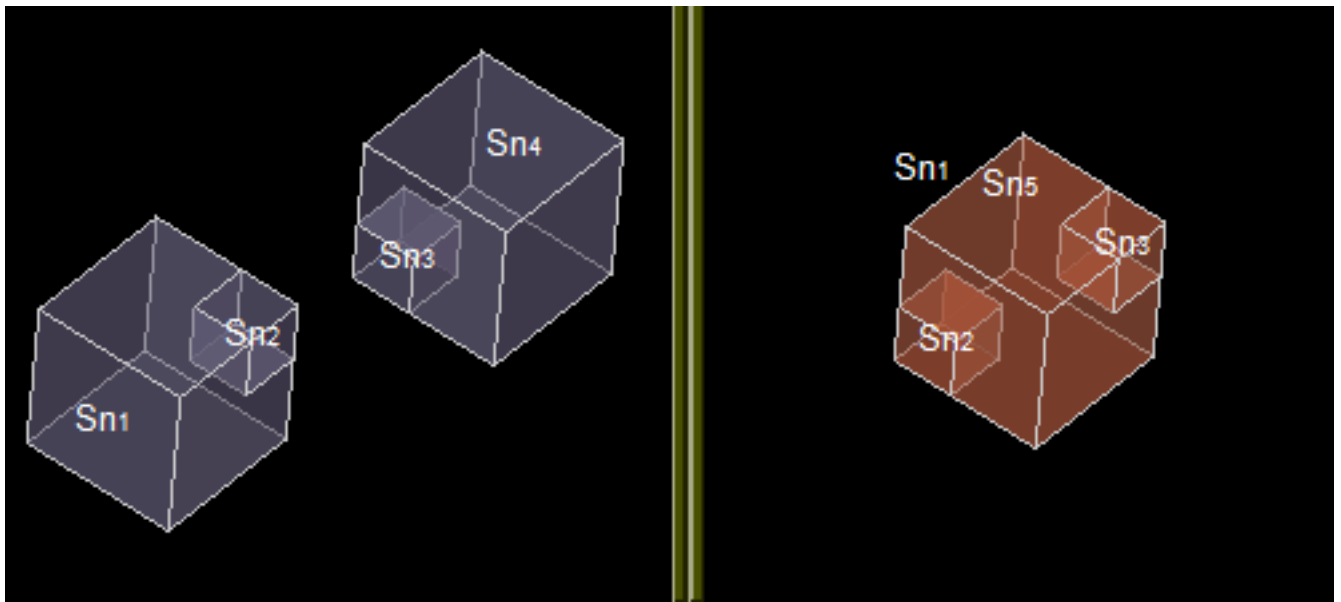


Figure 32: A compound and a solid: results

## 7.4 Class BOPAlgo\_Builder

GFA is implemented in the class *BOPAlgo\_Builder*.

### 7.4.1 Fields

The main fields of the class are described in the Table:

Name	Contents
<i>myPaveFiller</i>	Pointer to the <i>BOPAlgo_PaveFiller</i> object
<i>myDS</i>	Pointer to the <i>BOPDS_DS</i> object
<i>myContext</i>	Pointer to the intersection Context
<i>myImages</i>	The Map between the source shape and its images
<i>myShapesSD</i>	The Map between the source shape (or split part of source shape) and the shape (or part of shape) that will be used in result due to same domain property.

### 7.4.2 Initialization

The input data for this step is a *BOPAlgo\_PaveFiller* object (in terms of Intersection) at the state after Processing of degenerated edges with the corresponding DS.

No	Contents	Implementation
1	Check the readiness of the DS and <i>BOPAlgo_PaveFiller</i> .	<i>BOPAlgo_Builder::CheckData()</i>
2	Build an empty result of type Compound.	<i>BOPAlgo_Builder::Prepare()</i>

### 7.4.3 Build Images for Vertices

The input data for this step is *BOPAlgo\_Builder* object after Initialisation.

No	Contents	Implementation
1	Fill <i>myShapesSD</i> by SD vertices using the information from the DS.	<i>BOPAlgo_Builder::FillImages-Vertices()</i>

#### 7.4.4 Build Result of Type Vertex

The input data for this step is *BOPAlgo\_Builder* object after building images for vertices and *Type*, which is the shape type (*TopAbs\_VERTEX*).

No	Contents	Implementation
1	For the arguments of type <i>Type</i> . If there is an image for the argument: add the image to the result. If there is no image for the argument: add the argument to the result.	<i>BOPAlgo_Builder::BuildResult()</i>

#### 7.4.5 Build Images for Edges

The input data for this step is *BOPAlgo\_Builder* object after building result of type vertex.

No	Contents	Implementation
1	For all pave blocks in the DS. Fill <i>myImages</i> for the original edge <i>E</i> by split edges <i>ESPi</i> from pave blocks. In case of common blocks on edges, use edge <i>ESPSDj</i> that corresponds to the leading pave block and fill <i>myShapesSD</i> by the pairs <i>ESPi/ESPSDj</i> .	<i>BOPAlgo_Builder::FillImages-Edges()</i>

#### 7.4.6 Build Result of Type Edge

This step is the same as Building Result of Type Vertex, but for the type *Edge*.

#### 7.4.7 Build Images for Wires

The input data for this step is:

- *BOPAlgo\_Builder* object after building result of type *Edge*;
- Original Shape - Wire
- *Type* - the shape type (*TopAbs\_WIRE*).

No	Contents	Implementation
1	For all arguments of the type <i>Type</i> . Create a container C of the type <i>Type</i> .	<i>BOPAlgo_Builder::FillImages-Containers()</i>
2	Add to C the images or non-split parts of the <i>Original Shape</i> , taking into account its orientation.	<i>BOPAlgo_Builder::FillImages-Containers()</i> <i>BOPTools_Tools::IsSplitTo-Reverse()</i>

3	Fill <i>myImages</i> for the <i>Original Shape</i> by the information above.	<i>BOPAlgo_Builder::FillImagesContainers()</i>
---	--	--

#### 7.4.8 Build Result of Type Wire

This step is the same as Building Result of Type Vertex but for the type *Wire*.

#### 7.4.9 Build Images for Faces

The input data for this step is *BOPAlgo\_Builder* object after building result of type *Wire*.

No	Contents	Implementation
1	Build Split Faces for all interfered DS shapes <i>Fi</i> of type <i>FACE</i> .	
1.1	Collect all edges or their images of <i>Fi(ESPIj)</i> .	<i>BOPAlgo_Builder::BuildSplitFaces()</i>
1.2	Impart to <i>ESPIj</i> the orientation to be coherent with the original one.	<i>BOPAlgo_Builder::BuildSplitFaces()</i>
1.3	Collect all section edges <i>SEk</i> for <i>Fi</i> .	<i>BOPAlgo_Builder::BuildSplitFaces()</i>
1.4	Build split faces for <i>Fi</i> ( <i>Fi1</i> , <i>Fi2</i> ... <i>FiNbSp</i> ), where <i>NbSp</i> is the number of split parts (see Building faces from a set of edges for more details).	<i>BOPAlgo_BuilderFace</i>
1.5	Impart to ( <i>Fi1</i> , <i>Fi2</i> ... <i>FiNbSp</i> ) the orientation coherent with the original face <i>Fi</i> .	<i>BOPAlgo_Builder::BuildSplitFaces()</i>
1.6	Fill the map <i>mySplits</i> with <i>Fi/(Fi1</i> , <i>Fi2</i> ... <i>FiNbSp</i> )	<i>BOPAlgo_Builder::BuildSplitFaces()</i>
2	Fill Same Domain faces	<i>BOPAlgo_Builder::FillSameDomainFaces</i>
2.1	Find and collect in the contents of <i>mySplits</i> the pairs of same domain split faces ( <i>Fij</i> , <i>Fkl</i> ) <i>m</i> , where <i>m</i> is the number of pairs.	<i>BOPAlgo_Builder::FillSameDomainFaces</i> <i>BOPTools_Tools::AreFacesSameDomain()</i>
2.2	Compute the connexity chains 1) of same domain faces ( <i>F1C</i> , <i>F2C</i> ... <i>FnC</i> ) <i>k</i> , <i>C</i> =0, 1... <i>nCs</i> , where <i>nCs</i> is the number of connexity chains.	<i>BOPAlgo_Builder::FillSameDomainFaces()</i>
2.3	Fill <i>myShapesSD</i> using the chains ( <i>F1C</i> , <i>F2C</i> ... <i>FnC</i> ) <i>k</i>	<i>BOPAlgo_Builder::FillSameDomainFaces()</i>
2.4	Add internal vertices to split faces.	<i>BOPAlgo_Builder::FillSameDomainFaces()</i>
2.5	Fill <i>myImages</i> using <i>myShapesSD</i> and <i>mySplits</i> .	<i>BOPAlgo_Builder::FillSameDomainFaces()</i>

The example of chains of same domain faces is given in the image:

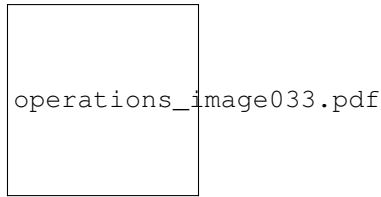


Figure 33: Chains of same domain faces

- The pairs of same domain faces are:  $(F11, F21)$ ,  $(F22, F31)$ ,  $(F41, F51)$ ,  $(F41, F6)$  and  $(F51, F6)$ .
- The pairs produce the three chains:  $(F11, F21)$ ,  $(F22, F31)$  and  $(F41, F51, F6)$ .

#### 7.4.10 Build Result of Type Face

This step is the same as Building Result of Type Vertex but for the type *Face*.

#### 7.4.11 Build Images for Shells

The input data for this step is:

- *BOPAlgo\_Builder* object after building result of type face;
- *Original Shape* - Shell;
- *Type* - the type of the shape (*TopAbs\_SHELL*).

The procedure is the same as for building images for wires.

#### 7.4.12 Build Result of Type Shell

This step is the same as Building Result of Type Vertex but for the type *Shell*.

#### 7.4.13 Build Images for Solids

The input data for this step is *BOPAlgo\_Builder* object after building result of type *Shell*.

The following procedure is executed for all interfered DS shapes *Si* of type *SOLID*.

No	Contents	Implementation
1	Collect all images or non-split parts for all faces ( <i>FSPij</i> ) that have 3D state <i>In Si</i> .	<i>BOPAlgo_Builder::FillIn3DParts ()</i>
2	Collect all images or non-split parts for all faces of <i>Si</i>	<i>BOPAlgo_Builder::BuildSplitSolids()</i>
3	Build split solids for <i>Si</i> -> ( <i>Si1</i> , <i>Si2</i> ... <i>SiNbSp</i> ), where <i>NbSp</i> is the number of split parts (see Building faces from a set of edges for more details)	<i>BOPAlgo_BuilderSolid</i>

4	Fill the map Same Domain solids <i>myShapesSD</i>	<i>BOPAlgo_Builder::BuildSplitSolids()</i>
5	Fill the map <i>myImages</i>	<i>BOPAlgo_Builder::BuildSplitSolids()</i>
6	Add internal vertices to split solids	<i>BOPAlgo_Builder::FillInternalShapes()</i>

#### 7.4.14 Build Result of Type Solid

This step is the same as Building Result of Type Vertex, but for the type Solid.

#### 7.4.15 Build Images for Type CompSolid

The input data for this step is:

- *BOPAlgo\_Builder* object after building result of type solid;
- *Original Shape* - Compsolid;
- *Type* - the type of the shape (*TopAbs\_COMPSOLID*).

The procedure is the same as for building images for wires.

#### 7.4.16 Build Result of Type Compsolid

This step is the same as Building Result of Type Vertex, but for the type Compsolid.

#### 7.4.17 Build Images for Compounds

The input data for this step is as follows:

- *BOPAlgo\_Builder* object after building results of type compsolid;
- *Original Shape* - Compound;
- *Type* - the type of the shape (*TopAbs\_COMPOUND*).

The procedure is the same as for building images for wires.

#### 7.4.18 Build Result of Type Compound

This step is the same as Building Result of Type Vertex, but for the type Compound.

#### 7.4.19 Post-Processing

The purpose of the step is to correct tolerances of the result to provide its validity in terms of *BRepCheck\_Analyzer*.

The input data for this step is a *BOPAlgo\_Builder* object after building result of type compound.

No	Contents	Implementation
1	Correct tolerances of vertices on curves	<i>BOPTools_Tools::CorrectPointOnCurve()</i>

2	Correct tolerances of edges on faces	<i>BOPTools_Tools::CorrectCurveOnSurface()</i>
---	--------------------------------------	--

## 8 Boolean Operations Algorithm

### 8.1 Arguments

- The arguments of BOA are shapes in terms of *TopoDS\_Shape*. The main requirements for the arguments are described in Algorithms
- There are only two arguments in BOA Operators:
  - Object (*S1*);
  - Tool (*S2*).
- Each argument should contain objects of a single dimension (see the Table).

No	Type of Argument	Index of Type	Dimension
1	COMPOUND	0	One of 0, 1, 2, 3
2	COMPSOLID	1	3
3	SOLID	2	3
4	SHELL	3	2
5	FACE	4	2
6	WIRE	5	1
7	EDGE	6	1
8	VERTEX	7	0

- For Boolean operation Fuse the arguments should have equal dimensions.
- For Boolean operation Cut the dimension of *S1* should not be less then the dimension of *S2*.

### 8.2 Results

During the operation the argument *Si* can be spited into parts *Si1*, *Si2* ... *Si1NbSp*, where *NbSp* is the number of parts. The set (*Si1*, *Si2* ... *Si1NbSp*) is an image of the argument *Si*.

### 8.3 General Rules

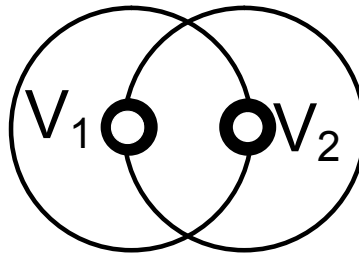
- The result of the BOA operation is a compound (if defined). Each sub-shape of the compound corresponds to the concrete argument *S1* and *S2* and has shared sub-shapes in accordance with interferences between the arguments.
- The content of the result depends on the type of the operation (Common, Fuse, Cut12, Cut21) and the dimensions of the arguments.
- The result of the operation Fuse is defined for arguments *S1* and *S2* that have the same dimension value :  $Dim(S1)=Dim(S2)$ . If the arguments have different dimension values the result of the operation Fuse is not defined. The dimension of the result is equal to the dimension of the arguments. For example, it is impossible to fuse an edge and a face.
- The result of the operation Fuse for arguments *S1* and *S2* contains the parts of arguments that have states **OUT** relative to the opposite arguments.
- The result of the operation Fuse for arguments *S1* and *S2* having dimension value 3 (Solids) is refined by removing all possible internal faces to provide minimal number of solids.
- The result of the operation Common for arguments *S1* and *S2* is defined for all values of the dimensions of the arguments. The dimension of the result is equal to the lower dimension of the arguments. For example, the result of the operation Common between edges can not be a vertex.
- The result of the operation Common for the arguments *S1* and *S2* contains the parts of the argument that have states **IN** and **ON** relative to the opposite argument.

- The result of the operation  $Cut12[Cut21]$  is defined for arguments  $S1$  and  $S2$  that have values of dimensions  $Dim(S1)Dim(S2)$   $[Dim(S1)Dim(S2)]$ . The dimension of the result is equal to the lower dimension of the arguments. The result of the operation  $Cut12[Cut21]$  is not defined for other cases. For example, it is impossible to cut a solid from an edge, because such shape type as a solid without an edge is not defined.
- The result of the operation  $Cut12$  for arguments  $S1$  and  $S2$  contains the parts of argument  $S1$  that have state **OUT** relative to the opposite argument  $S2$ .
- The result of the operation  $Cut21$  for arguments  $S1$  and  $S2$  contains the parts of argument  $S2$  that have state **OUT** relative to the opposite argument  $S1$ .

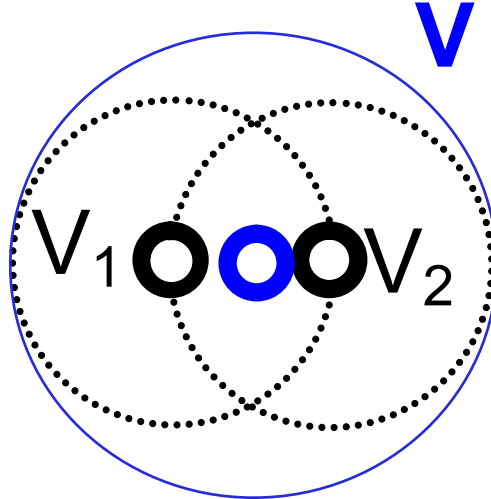
## 8.4 Examples

### 8.4.1 Case 1: Two Vertices

Let us consider two interfering vertices  $V1$  and  $V2$ :



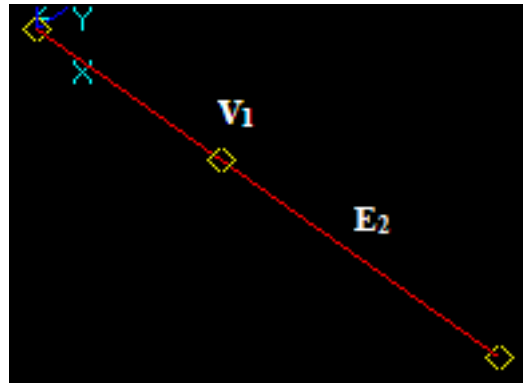
- The result of *Fuse* operation is the compound that contains new vertex  $V$ .



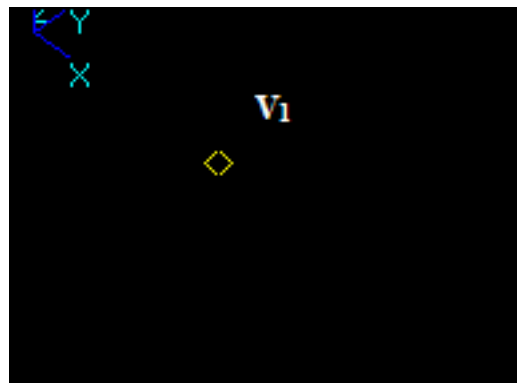
- The result of *Common* operation is a compound containing new vertex  $V$ . This compound is identical to the result of *Fuse* operation.
- The result of *Cut12* operation is an empty compound.
- The result of *Cut21* operation is an empty compound.

### 8.4.2 Case 2: A Vertex and an Edge

Let us consider vertex  $V1$  and the edge  $E2$ , that intersect in a 3D point:



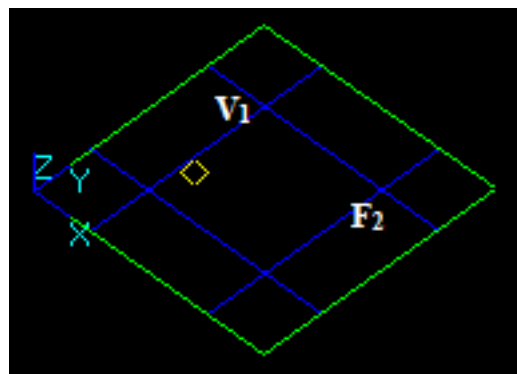
- The result of *Fuse* operation is result is not defined because the dimension of the vertex (0) is not equal to the dimension of the edge (1).
- The result of *Common* operation is a compound containing a new vertex  $V_7$  as the argument  $V_7$  has a common part with edge  $E_2$ .



- The result of *Cut12* operation is an empty compound.
- The result of *Cut21* operation is not defined because the dimension of the vertex (0) is less than the dimension of the edge (1).

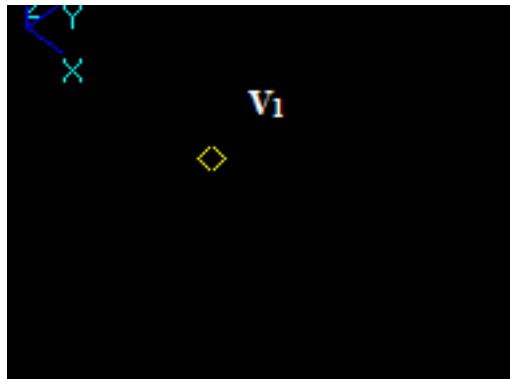
#### 8.4.3 Case 3: A Vertex and a Face

Let us consider vertex  $V_1$  and face  $F_2$ , that intersect in a 3D point:



- The result of *Fuse* operation is not defined because the dimension of the vertex (0) is not equal to the dimension of the face (2).

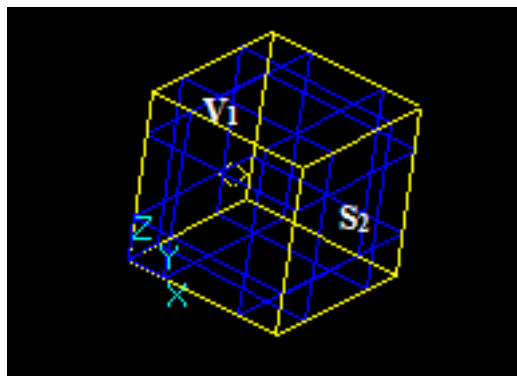
- The result of *Common* operation is a compound containing a new vertex  $V_7$  as the argument  $V_7$  has a common part with face  $F_2$ .



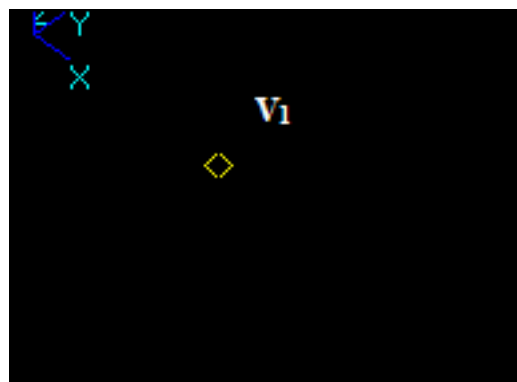
- The result of *Cut12* operation is an empty compound.
- The result of *Cut21* operation is not defined because the dimension of the vertex (0) is less than the dimension of the face (2).

#### 8.4.4 Case 4: A Vertex and a Solid

Let us consider vertex  $V_1$  and solid  $S_2$ , that intersect in a 3D point:



- The result of *Fuse* operation is not defined because the dimension of the vertex (0) is not equal to the dimension of the solid (3).
- The result of *Common* operation is a compound containing a new vertex  $V_7$  as the argument  $V_7$  has a common part with solid  $S_2$ .

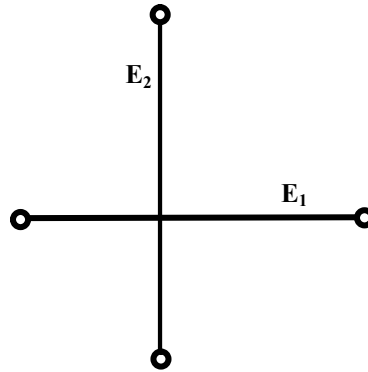


- The result of *Cut12* operation is an empty compound.

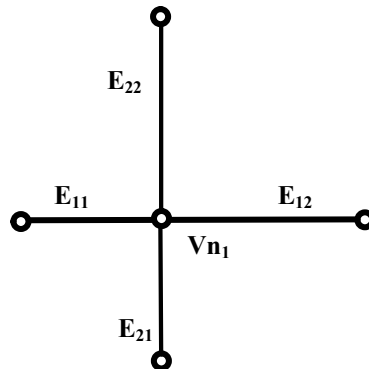
- The result of *Cut21* operation is not defined because the dimension of the vertex (0) is less than the dimension of the solid (3).

#### 8.4.5 Case 5: Two edges intersecting at one point

Let us consider edges  $E1$  and  $E2$ , that intersect in a 3D point:

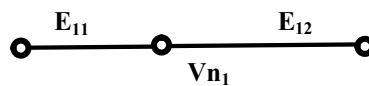


- The result of *Fuse* operation is a compound containing split parts of arguments i.e. 4 new edges  $E11$ ,  $E12$ ,  $E21$ , and  $E22$ . These edges have one shared vertex  $Vn1$ . In this case:
  - argument edge  $E1$  has resulting split edges  $E11$  and  $E12$  (image of  $E1$ );
  - argument edge  $E2$  has resulting split edges  $E21$  and  $E22$  (image of  $E2$ ).



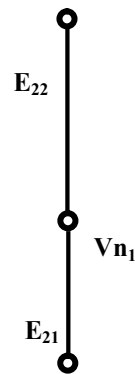
- The result of *Common* operation is an empty compound because the dimension (0) of the common part between the edges (vertex) is less than the dimension of the arguments (1).
- The result of *Cut12* operation is a compound containing split parts of the argument  $E1$ , i.e. 2 new edges  $E11$  and  $E12$ . These edges have one shared vertex  $Vn1$ .

In this case the argument edge  $E1$  has resulting split edges  $E11$  and  $E12$  (image of  $*E1$ ).



- The result of *Cut21* operation is a compound containing split parts of the argument  $E2$ , i.e. 2 new edges  $E21$  and  $E12$ . These edges have one shared vertex  $Vn1$ .

In this case the argument edge  $E2$  has resulting split edges  $E21$  and  $E22$  (image of  $E2$ ).

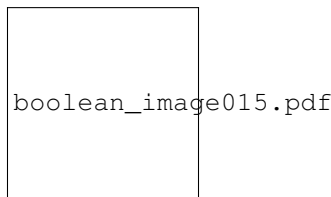


#### 8.4.6 Case 6: Two edges having a common block

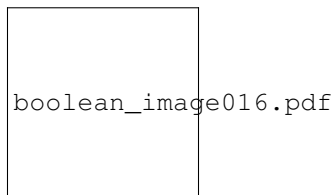
Let us consider edges  $E1$  and  $E2$ , that have a common block:



- The result of *Fuse* operation is a compound containing split parts of arguments i.e. 3 new edges  $E11$ ,  $E12$  and  $*E22$ . These edges have two shared vertices. In this case:
  - argument edge  $E1$  has resulting split edges  $E11$  and  $E12$  (image of  $E1$ );
  - argument edge  $E2$  has resulting split edges  $E21$  and  $E22$  (image of  $E2$ );
  - edge  $E12$  is common for the images of  $E1$  and  $E2$ .



- The result of *Common* operation is an empty compound containing split parts of arguments i.e. 1 new edge  $E12$ . In this case edge  $E12$  is common for the images of  $E1$  and  $E2$ . The common part between the edges (edge) has the same dimension (1) as the dimension of the arguments (1).



- The result of *Cut12* operation is a compound containing a split part of argument  $E1$ , i.e. new edge  $E11$ .

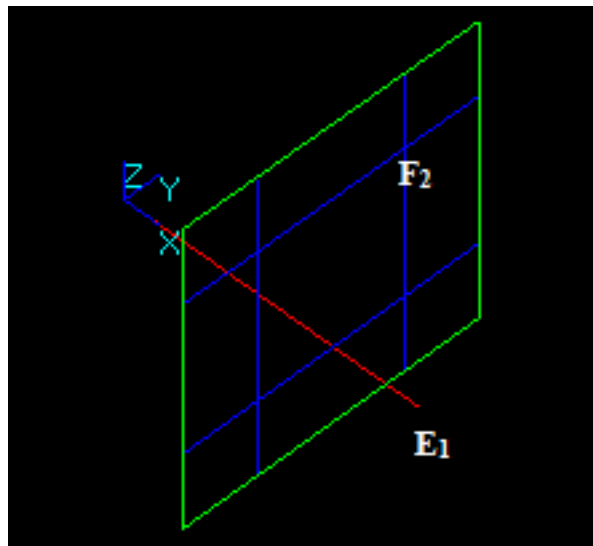


- The result of *Cut21* operation is a compound containing a split part of argument  $E2$ , i.e. new edge  $E22$ .



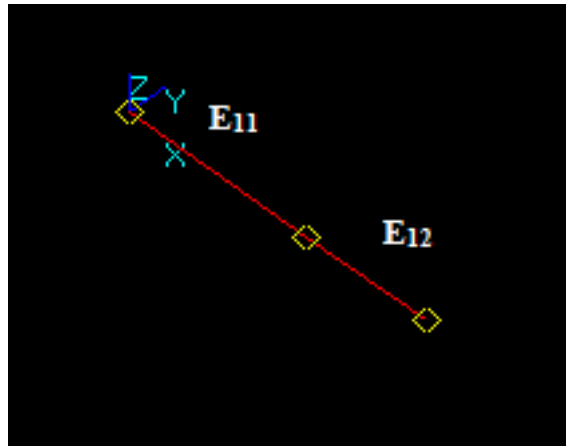
#### 8.4.7 Case 7: An Edge and a Face intersecting at a point

Let us consider edge  $E1$  and face  $F2$ , that intersect at a 3D point:



- The result of *Fuse* operation is not defined because the dimension of the edge (1) is not equal to the dimension of the face (2).
- The result of *Common* operation is an empty compound because the dimension (0) of the common part between the edge and face (vertex) is less than the dimension of the arguments (1).
- The result of *Cut12* operation is a compound containing split parts of the argument  $E1$ , i.e. 2 new edges  $E11$  and  $E12$ .

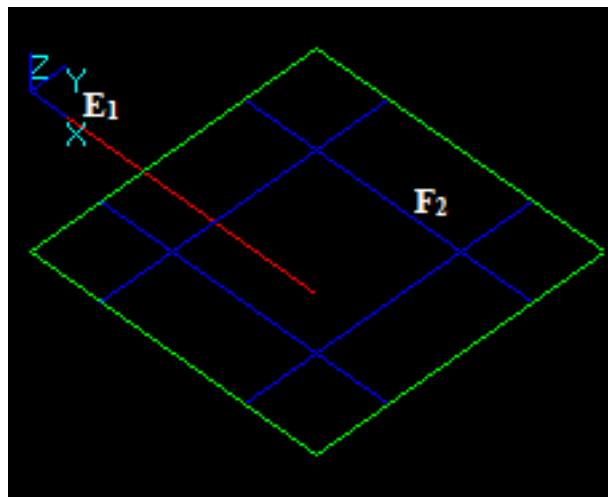
In this case the argument edge  $E1$  has no common parts with the face  $F2$  so the whole image of  $E1$  is in the result.



- The result of *Cut21* operation is not defined because the dimension of the edge (1) is less than the dimension of the face (2).

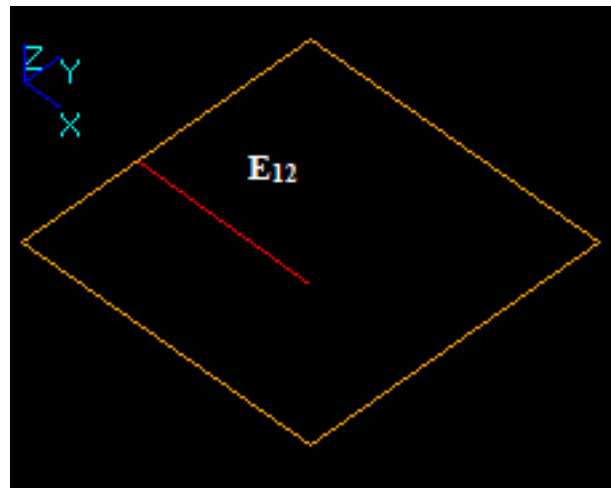
#### 8.4.8 Case 8: A Face and an Edge that have a common block

Let us consider edge  $E1$  and face  $F2$ , that have a common block:



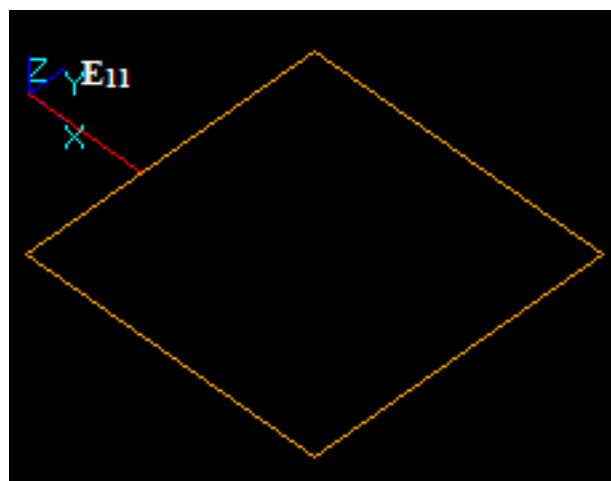
- The result of *Fuse* operation is not defined because the dimension of the edge (1) is not equal to the dimension of the face (2).
- The result of *Common* operation is a compound containing a split part of the argument  $E1$ , i.e. new edge  $E12$ .

In this case the argument edge  $E1$  has a common part with face  $F2$  so the corresponding part of the image of  $E1$  is in the result. The yellow square is not a part of the result. It only shows the place of  $F2$ .



- The result of *Cut12* operation is a compound containing split part of the argument  $E1$ , i.e. new edge  $E11$ .

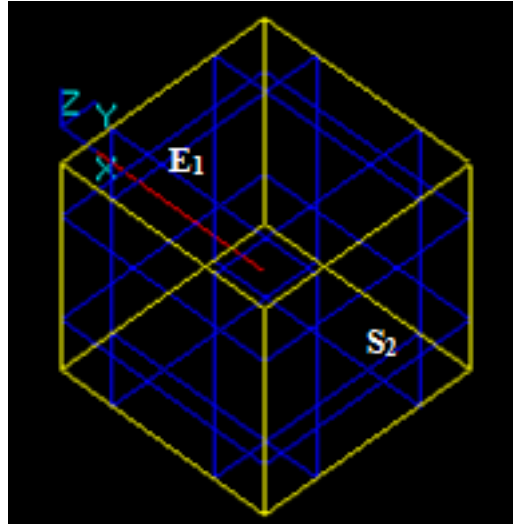
In this case the argument edge  $E1$  has a common part with face  $F2$  so the corresponding part is not included into the result. The yellow square is not a part of the result. It only shows the place of  $F2$ .



- The result of *Cut21* operation is not defined because the dimension of the edge (1) is less than the dimension of the face (2).

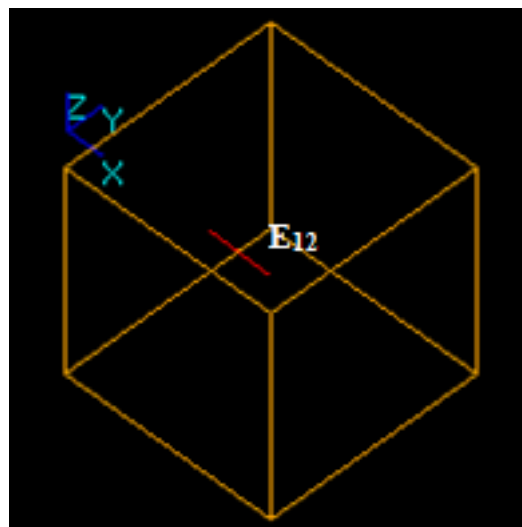
#### 8.4.9 Case 9: An Edge and a Solid intersecting at a point

Let us consider edge  $E1$  and solid  $S2$ , that intersect at a point:



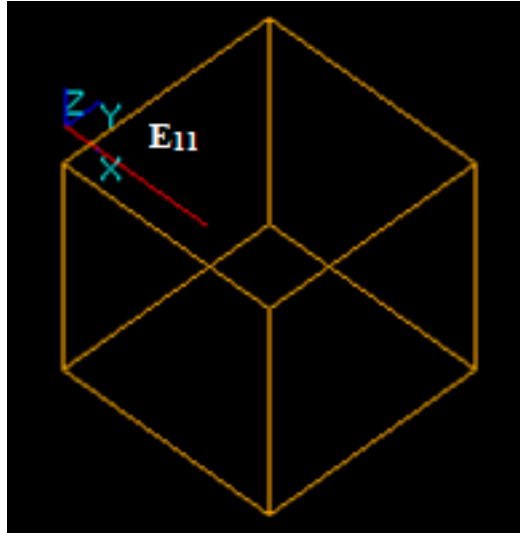
- The result of *Fuse* operation is not defined because the dimension of the edge (1) is not equal to the dimension of the solid (3).
- The result of *Common* operation is a compound containing a split part of the argument  $E1$ , i.e. new edge  $E12$ .

In this case the argument edge  $E1$  has a common part with solid  $S2$  so the corresponding part of the image of  $E1$  is in the result. The yellow square is not a part of the result. It only shows the place of  $S2$ .



- The result of *Cut12* operation is a compound containing split part of the argument  $E1$ , i.e. new edge  $E11$ .

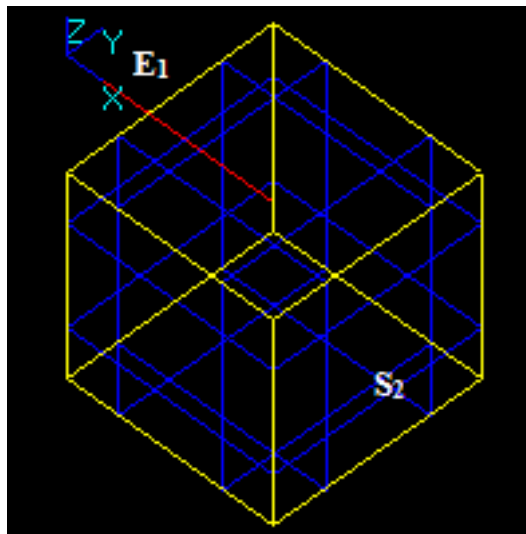
In this case the argument edge  $E1$  has a common part with solid  $S2$  so the corresponding part is not included into the result. The yellow square is not a part of the result. It only shows the place of  $S2$ .



- The result of *Cut21* operation is not defined because the dimension of the edge (1) is less than the dimension of the solid (3).

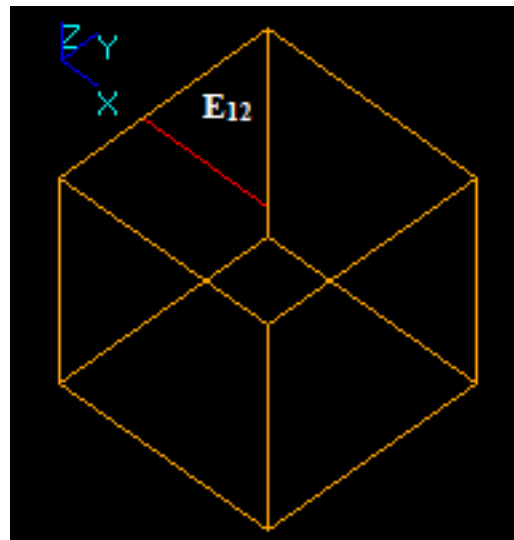
#### 8.4.10 Case 10: An Edge and a Solid that have a common block

Let us consider edge  $E1$  and solid  $S2$ , that have a common block:



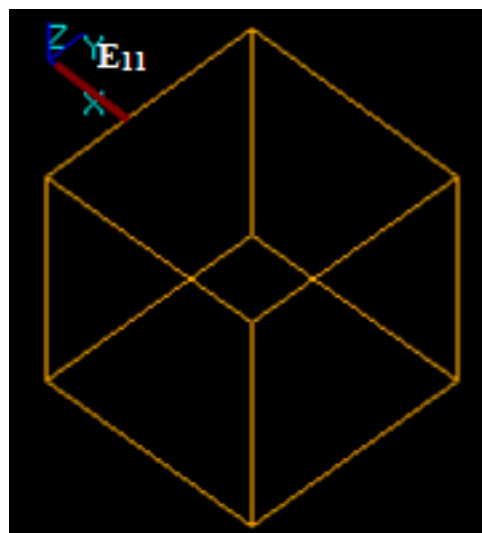
- The result of *Fuse* operation is not defined because the dimension of the edge (1) is not equal to the dimension of the solid (3).
- The result of *Common* operation is a compound containing a split part of the argument  $E1$ , i.e. new edge  $E12$ .

In this case the argument edge  $E1$  has a common part with solid  $S2$  so the corresponding part of the image of  $E1$  is in the result. The yellow square is not a part of the result. It only shows the place of  $S2$ .



- The result of *Cut12* operation is a compound containing split part of the argument  $E1$ , i.e. new edge  $E11$ .

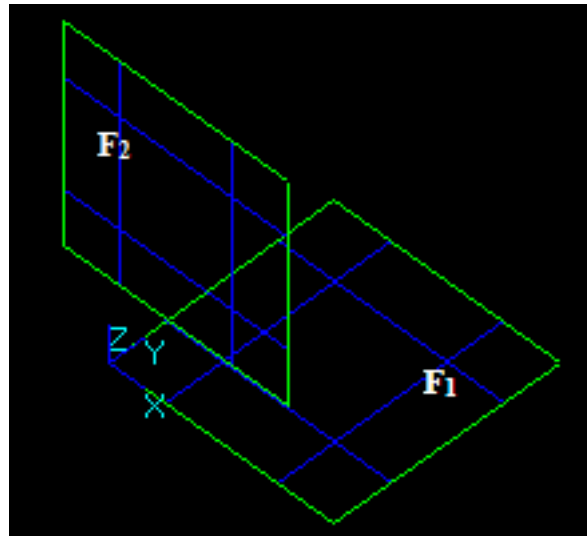
In this case the argument edge  $E1$  has a common part with solid  $S2$  so the corresponding part is not included into the result. The yellow square is not a part of the result. It only shows the place of  $S2$ .



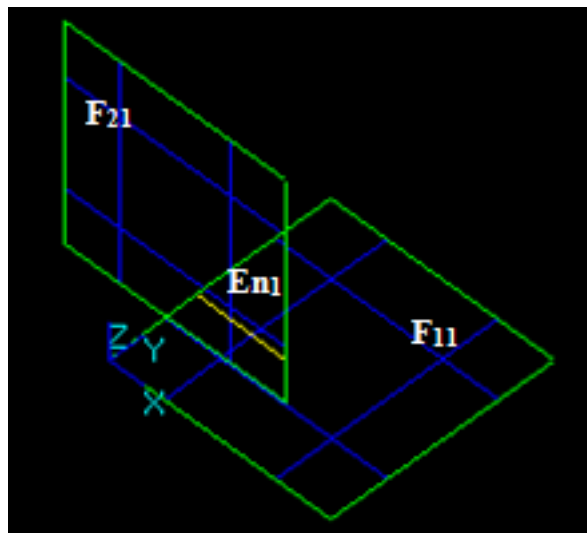
- The result of *Cut21* operation is not defined because the dimension of the edge (1) is less than the dimension of the solid (3).

#### 8.4.11 Case 11: Two intersecting faces

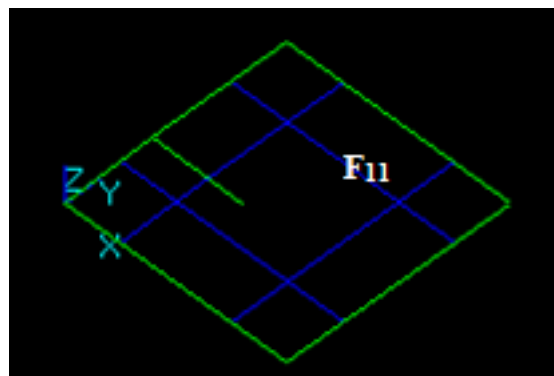
Let us consider two intersecting faces  $F1$  and  $F2$ :



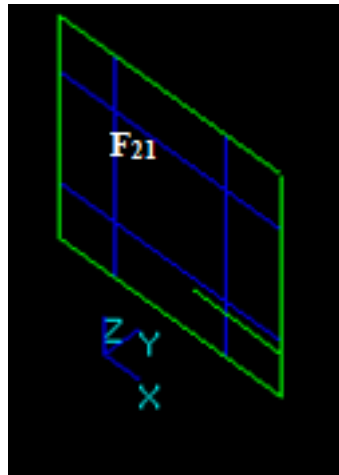
- The result of *Fuse* operation is an empty compound containing split parts of arguments i.e. 2 new faces  $F11$  and  $F21$ . These faces have one shared edge  $En1$ .



- The result of *Common* operation is an empty compound because the dimension (1) of the common part between  $F1$  and  $F2$  (edge) is less than the dimension of arguments (2).
- The result of *Cut12* operation is a compound containing split part of the argument  $F1$ , i.e. new face  $F11$ .

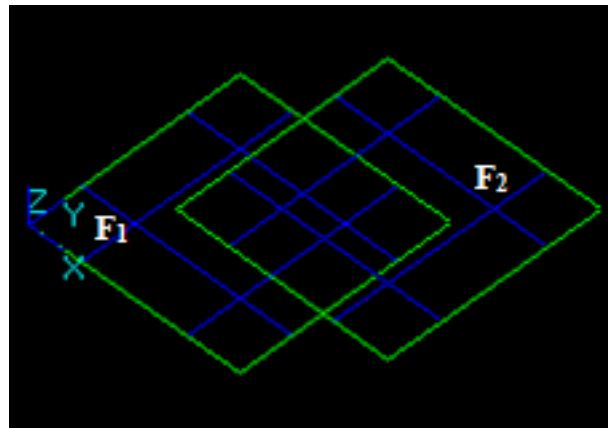


- The result of *Cut21* operation is a compound containing split parts of the argument  $F2$ , i.e. 1 new face  $F21$ .

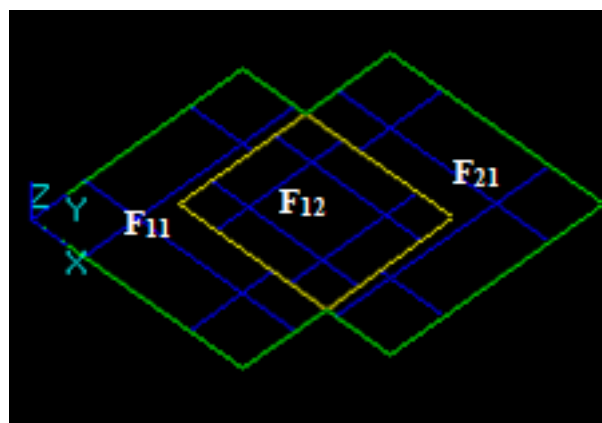


#### 8.4.12 Case 12: Two faces that have a common part

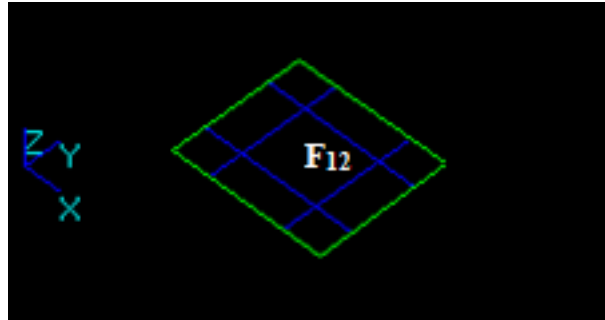
Let us consider two faces  $F1$  and  $F2$  that have a common planar part:



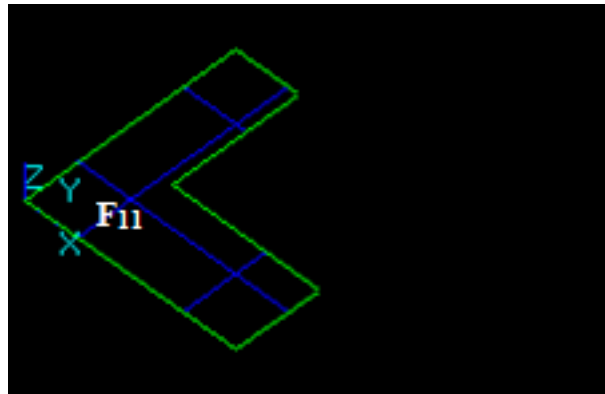
- The result of *Fuse* operation is a compound containing split parts of arguments, i.e. 3 new faces:  $F11$ ,  $F12$  and  $F22$ . These faces are shared through edges In this case:
  - the argument edge  $F1$  has resulting split faces  $F11$  and  $F12$  (image of  $F1$ )
  - the argument face  $F2$  has resulting split faces  $F12$  and  $F22$  (image of  $F2$ )
  - the face  $F12$  is common for the images of  $F1$  and  $F2$ .



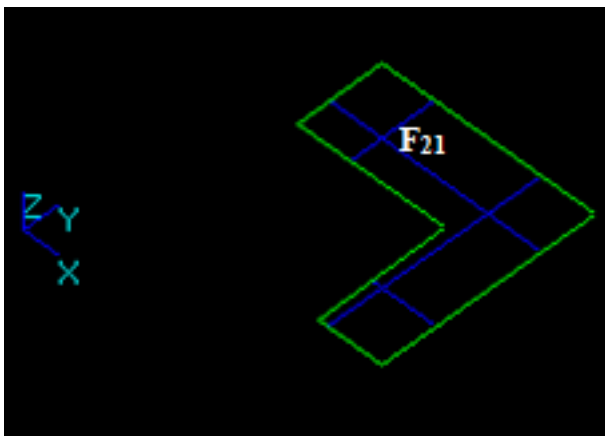
- The result of *Common* operation is a compound containing split parts of arguments i.e. 1 new face  $F_{12}$ . In this case: face  $F_{12}$  is common for the images of  $F_1$  and  $F_2$ . The common part between the faces (face) has the same dimension (2) as the dimension of the arguments (2).



- The result of *Cut12* operation is a compound containing split part of the argument  $F_1$ , i.e. new face  $F_{11}$ .

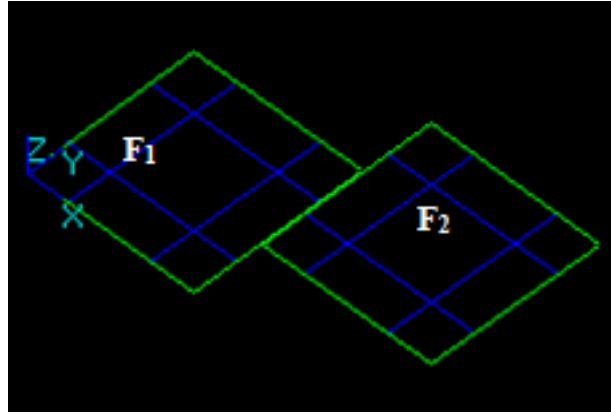


- The result of *Cut21* operation is a compound containing split parts of the argument  $F_2$ , i.e. 1 new face  $F_{21}$ .

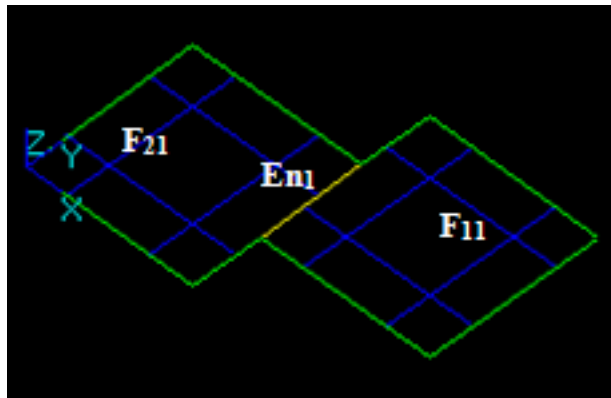


#### 8.4.13 Case 13: Two faces that have a common edge

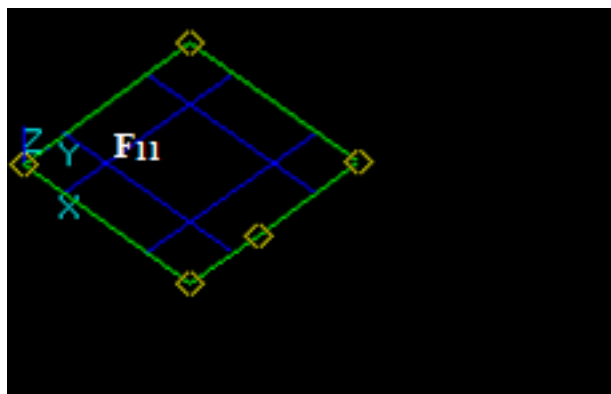
Let us consider two faces  $F_1$  and  $F_2$  that have a common edge:



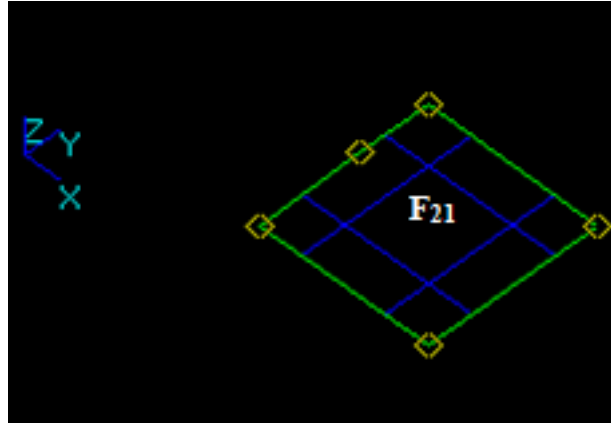
- The result of *Fuse* operation is a compound containing split parts of arguments, i.e. 2 new faces:  $F11$  and  $F21$ . These faces have one shared edge  $En1$ .



- The result of *Common* operation is an empty compound because the dimension (1) of the common part between  $F1$  and  $F2$  (edge) is less than the dimension of the arguments (2).
- The result of *Cut12* operation is a compound containing split part of the argument  $F1$ , i.e. new face  $F11$ . The vertices are shown just to clarify the fact that the edges are spitted.

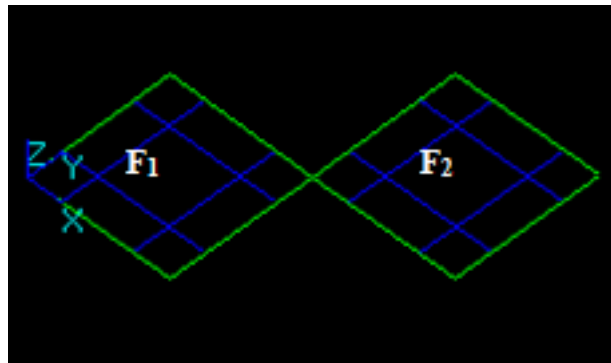


- The result of *Cut21* operation is a compound containing split parts of the argument  $F2$ , i.e. 1 new face  $F21$ . The vertices are shown just to clarify the fact that the edges are spitted.

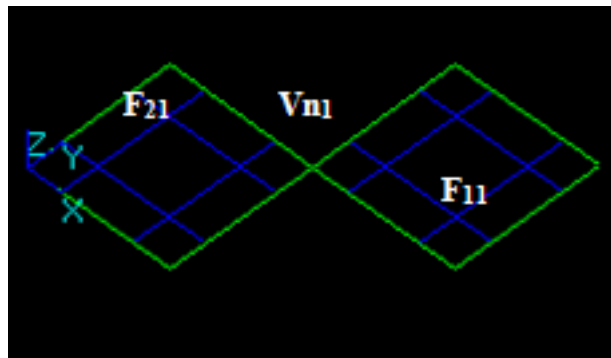


#### 8.4.14 Case 14: Two faces that have a common vertex

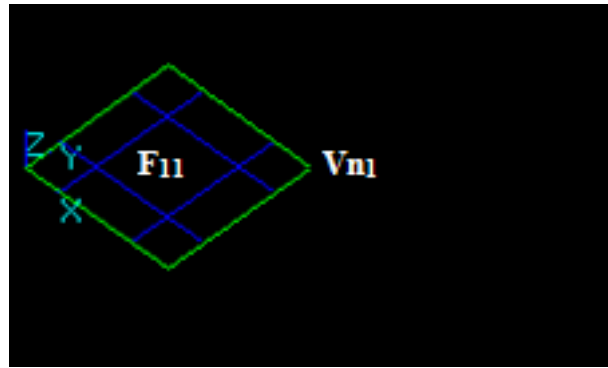
Let us consider two faces  $F1$  and  $F2$  that have a common vertex:



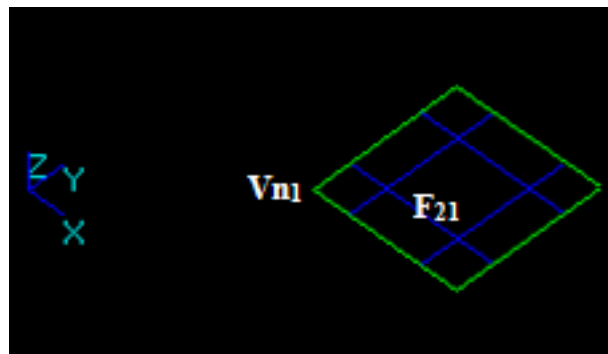
- The result of *Fuse* operation is a compound containing split parts of arguments, i.e. 2 new faces:  $F11$  and  $F21$ . These faces have one shared edge  $Vn1$ .



- The result of *Common* operation is an empty compound because the dimension (0) of the common part between  $F1$  and  $F2$  (vertex) is less than the dimension of the arguments (2)
- The result of *Cut12* operation is a compound containing split part of the argument  $F1$ , i.e. new face  $F11$ .

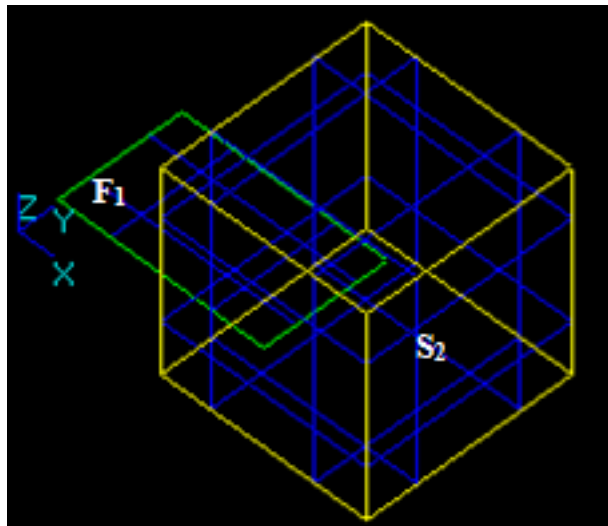


- The result of *Cut21* operation is a compound containing split parts of the argument  $F2$ , i.e. 1 new face  $F21$ .

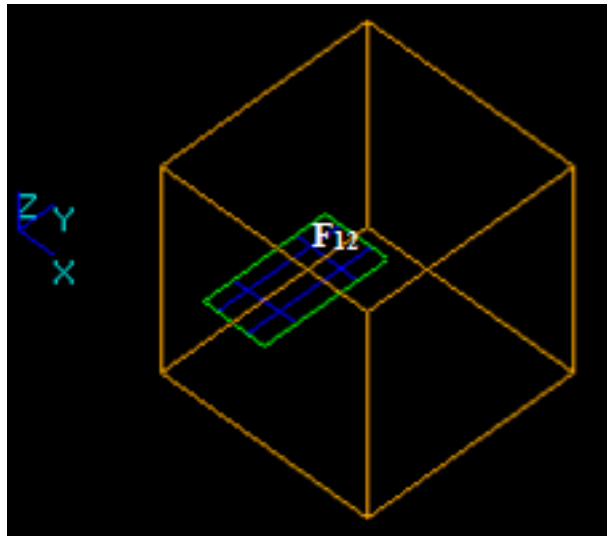


#### 8.4.15 Case 15: A Face and a Solid that have an intersection curve.

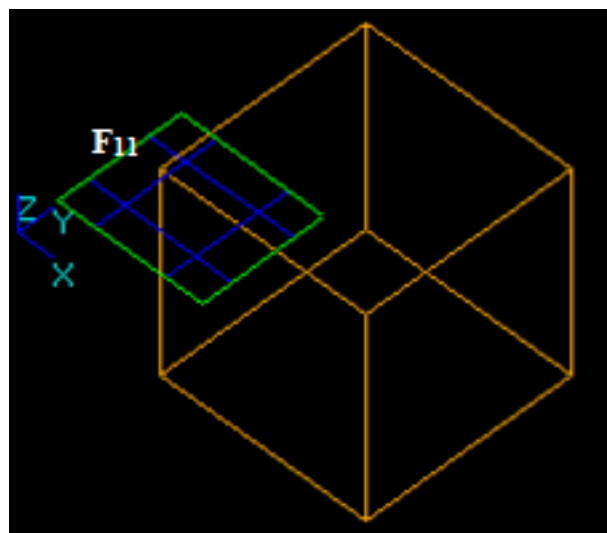
Let us consider face  $F1$  and solid  $S2$  that have an intersection curve:



- The result of *Fuse* operation is not defined because the dimension of the face (2) is not equal to the dimension of the solid (3).
- The result of *Common* operation is a compound contain split part of the argument  $F1$ . In this case the argument face  $F1$  has a common part with solid  $S2$ , so the corresponding part of the image of  $F1$  is in the result. The yellow contour is not a part of the result. It only shows the place of  $S2$ .



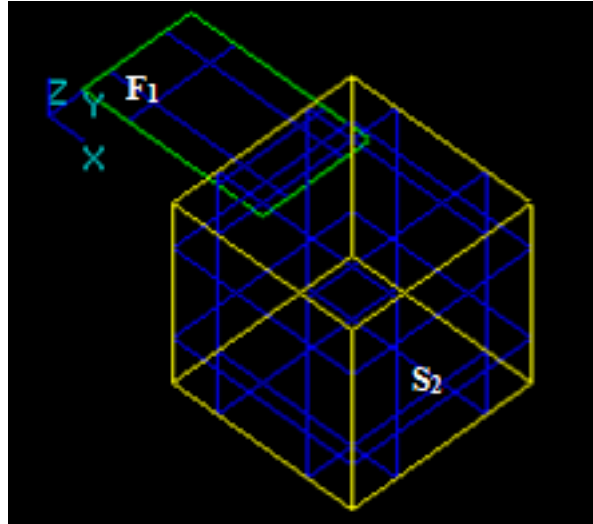
- The result of *Cut12* operation is a compound containing split part of the argument *F1*. In this case argument face *F1* has a common part with solid *S2* so the corresponding part is not included into the result. The yellow contour is not a part of the result. It only shows the place of *S2*.



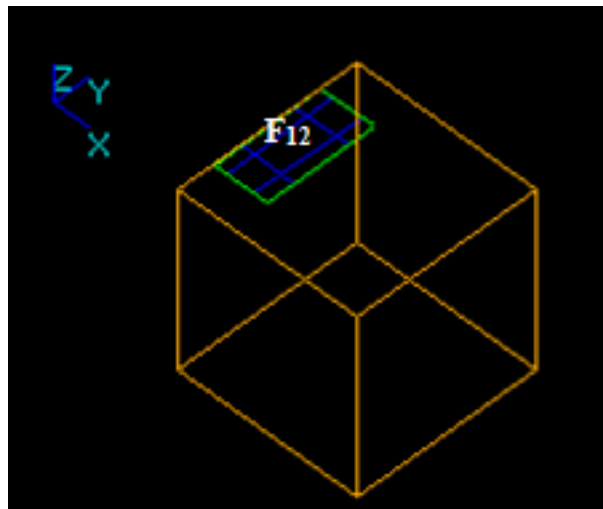
- The result of *Cut21* operation is not defined because the dimension of the face (2) is less than the dimension of the solid (3).

#### 8.4.16 Case 16: A Face and a Solid that have a common face.

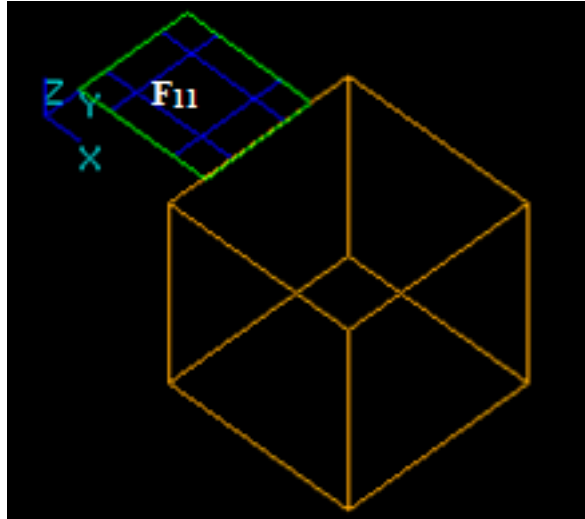
Let us consider face *F1* and solid *S2* that have a common part on face:



- The result of *Fuse* operation is not defined because the dimension of the face (2) is not equal to the dimension of the solid (3).
- The result of *Common* operation is a compound contain split part of the argument *F1*. In this case the argument face *F1* has a common part with solid *S2*, so the corresponding part of the image of *F1* is in the result. The yellow contour is not a part of the result. It only shows the place of *S2*.



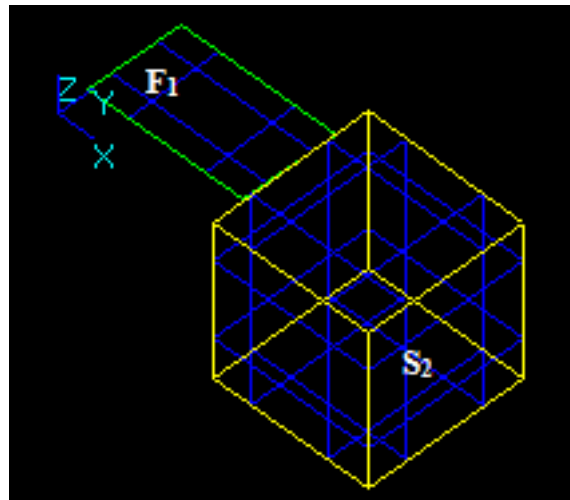
- The result of *Cut12* operation is a compound containing split part of the argument *F1*. In this case argument face *F1* has a common part with solid *S2* so the corresponding part is not included into the result. The yellow contour is not a part of the result. It only shows the place of *S2*.



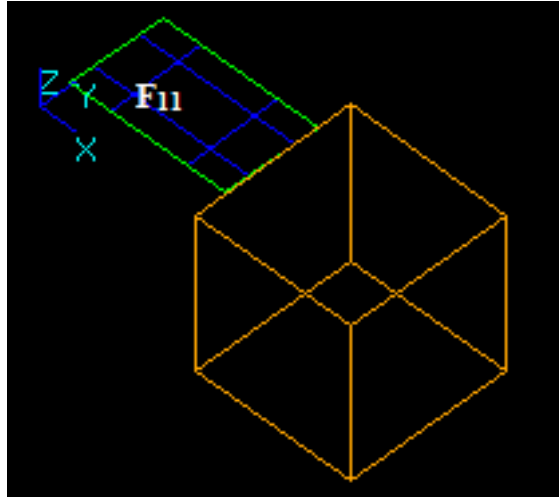
- The result of *Cut21* operation is not defined because the dimension of the face (2) is less than the dimension of the solid (3).

#### 8.4.17 Case 17: A Face and a Solid that have a common edge.

Let us consider face  $F1$  and solid  $S2$  that have a common part on edge:



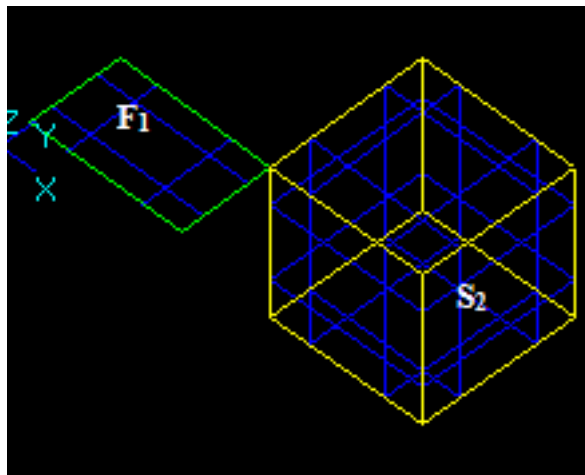
- The result of *Fuse* operation is not defined because the dimension of the face (2) is not equal to the dimension of the solid (3).
- The result of *Common* operation is an empty compound because the dimension (1) of the common part between  $F1$  and  $S2$  (edge) is less than the lower dimension of the arguments (2).
- The result of *Cut12* operation is a compound containing split part of the argument  $F1$ . In this case argument face  $F1$  has a common part with solid  $S2$  so the corresponding part is not included into the result. The yellow contour is not a part of the result. It only shows the place of  $S2$ .



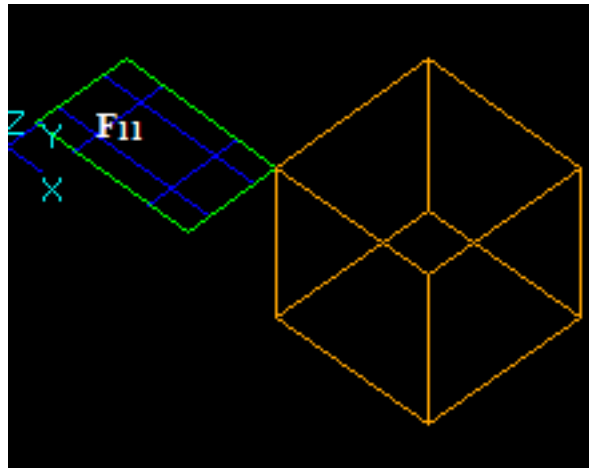
- The result of *Cut21* operation is not defined because the dimension of the face (2) is less than the dimension of the solid (3).

#### 8.4.18 Case 18: A Face and a Solid that have a common vertex.

Let us consider face  $F1$  and solid  $S2$  that have a common part on vertex:



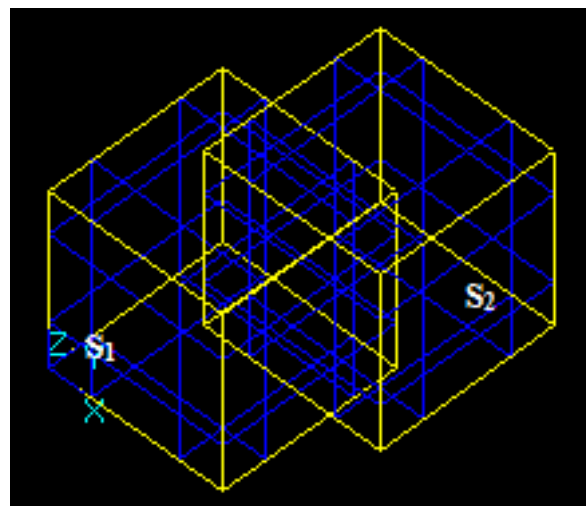
- The result of *Fuse* operation is not defined because the dimension of the face (2) is not equal to the dimension of the solid (3).
- The result of *Common* operation is an empty compound because the dimension (1) of the common part between  $F1$  and  $S2$  (vertex) is less than the lower dimension of the arguments (2).
- The result of *Cut12* operation is a compound containing split part of the argument  $F1$ . In this case argument face  $F1$  has a common part with solid  $S2$  so the corresponding part is not included into the result. The yellow contour is not a part of the result. It only shows the place of  $S2$ .



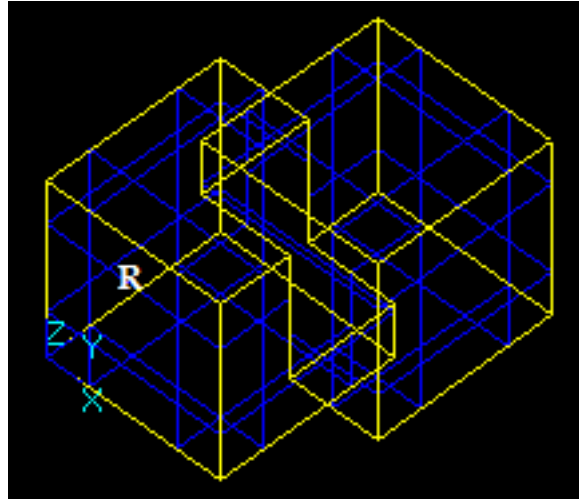
- The result of *Cut21* operation is not defined because the dimension of the face (2) is less than the dimension of the solid (3).

#### 8.4.19 Case 19: Two intersecting Solids.

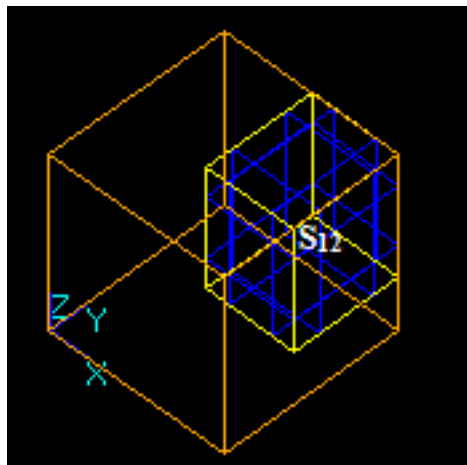
Let us consider two intersecting solids  $S1$  and  $S2$ :



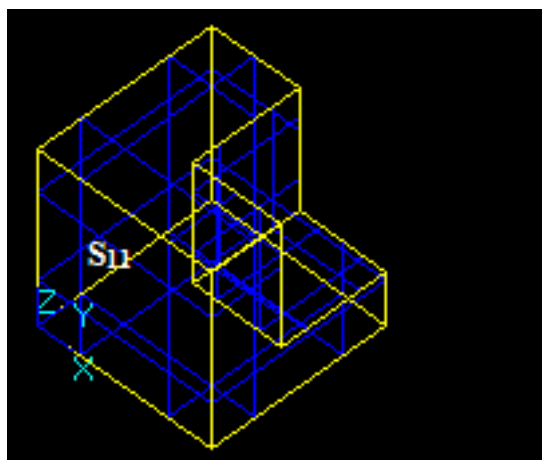
- The result of *Fuse* operation is a compound composed from the split parts of arguments  $S11$ ,  $S12$  and  $S22$   $*(Cut12, Common, Cut21)*$ . All inner webs are removed, so the result is one new solid  $R$ .



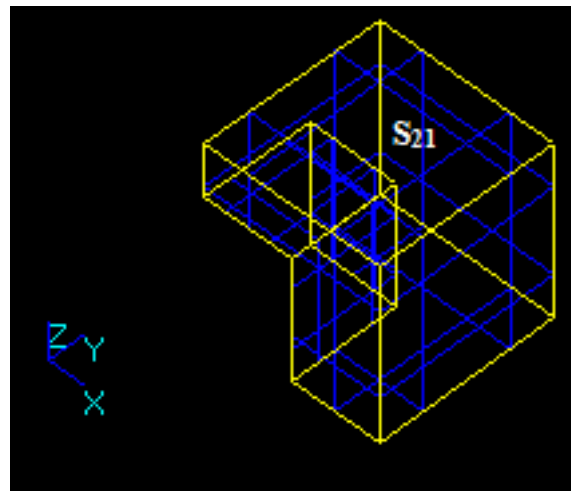
- The result of *Common* operation is a compound containing split parts of arguments i.e. one new solid *S12*. In this case solid *S12* is common for the images of *S1* and *S2*. The common part between the solids (solid) has the same dimension (3) as the dimension of the arguments (3). The yellow contour is not a part of the result. It only shows the place of *S1*.



- The result of *Cut12* operation is a compound containing split part of the argument *S1*, i.e. 1 new solid *S11*.

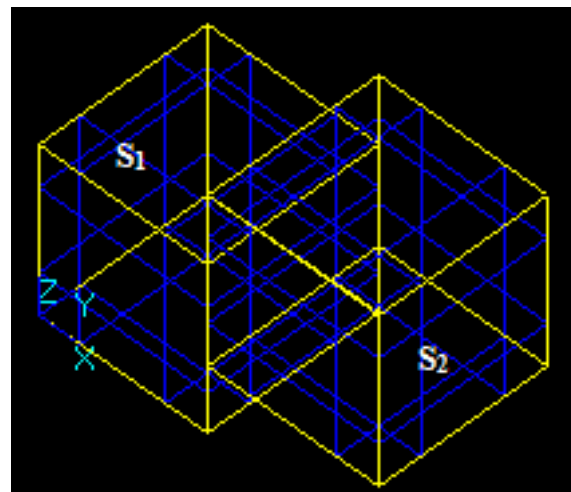


- The result of *Cut21* operation is a compound containing split part of the argument *S2*, i.e. 1 new solid *S21*.

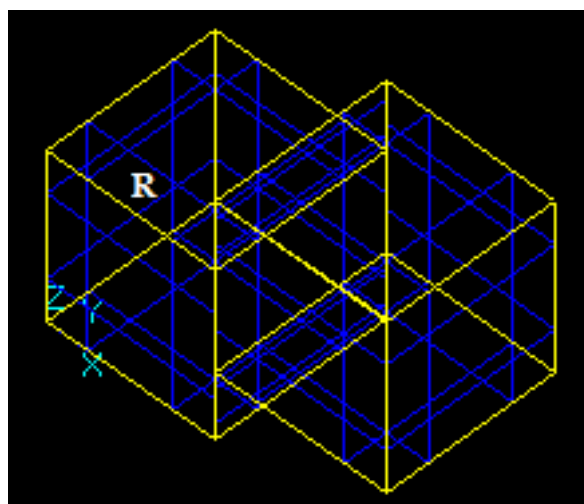


#### 8.4.20 Case 20: Two Solids that have a shared face.

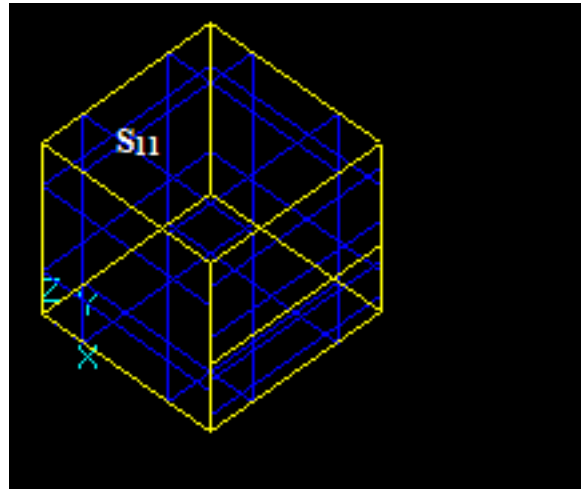
Let us consider two solids  $S_1$  and  $S_2$  that have a common part on face:



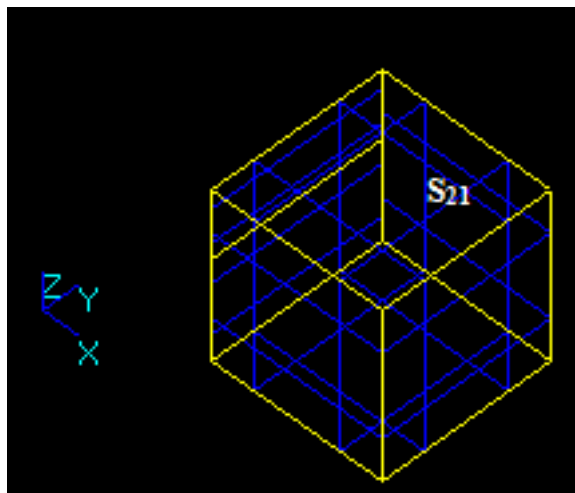
- The result of *Fuse* operation is a compound composed from the split parts of arguments  $S_{11}$ ,  $S_{12}$  and  $S_{22}$   $*(Cut_{12}, Common, Cut_{21})^*$ . All inner webs are removed, so the result is one new solid  $R$ .



- The result of *Common* operation is an empty compound because the dimension (2) of the common part between  $S1$  and  $S2$  (face) is less than the lower dimension of the arguments (3).
- The result of *Cut12* operation is a compound containing split part of the argument  $S1$ , i.e. 1 new solid  $S11$ .

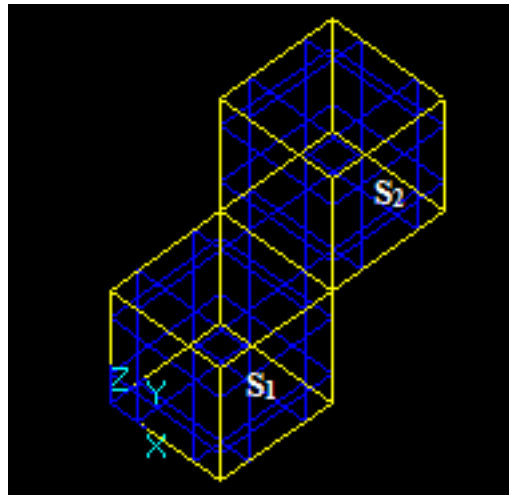


- The result of *Cut21* operation is a compound containing split part of the argument  $S2$ , i.e. 1 new solid  $S21$ .

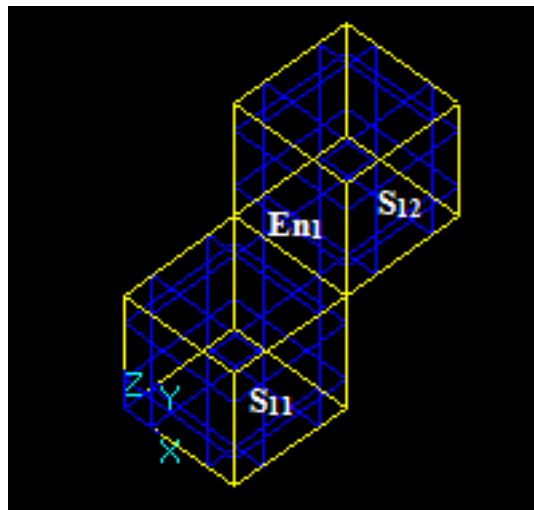


#### 8.4.21 Case 21: Two Solids that have a shared edge.

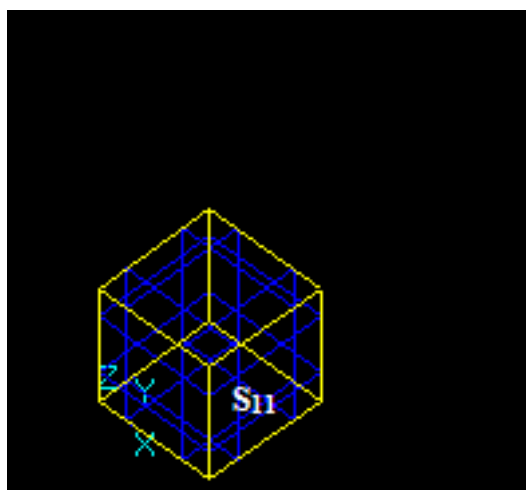
Let us consider two solids  $S1$  and  $S2$  that have a shared edge:



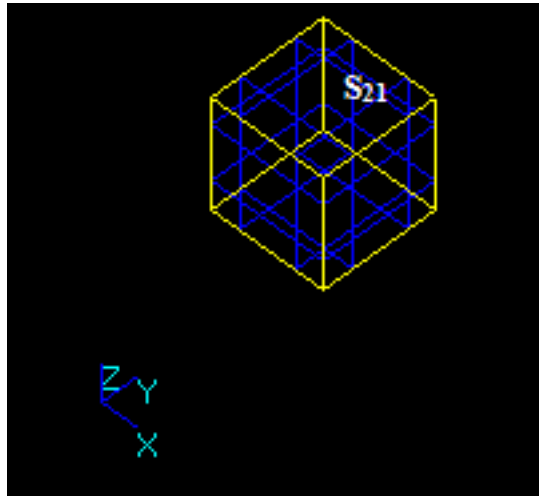
- The result of *Fuse* operation is a compound composed from the split parts of arguments i.e. 2 new solids  $S11$  and  $S21$ . These solids have one shared edge  $En1$ .



- The result of *Common* operation is an empty compound because the dimension (1) of the common part between  $S1$  and  $S2$  (edge) is less than the lower dimension of the arguments (3).
- The result of *Cut12* operation is a compound containing split part of the argument  $S1$ . In this case argument  $S1$  has a common part with solid  $S2$  so the corresponding part is not included into the result.

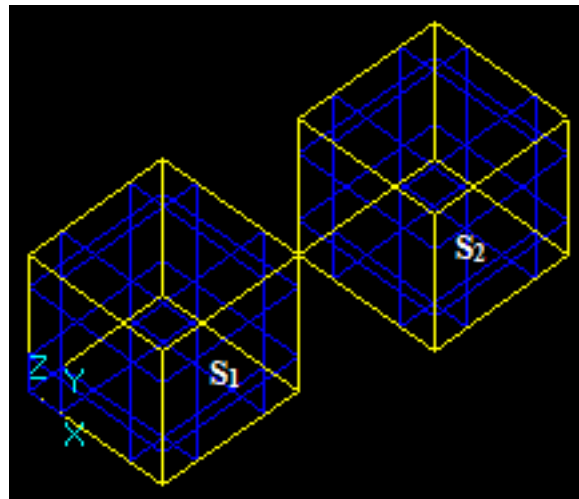


- The result of *Cut21* operation is a compound containing split part of the argument  $S2$ . In this case argument  $S2$  has a common part with solid  $S1$  so the corresponding part is not included into the result.

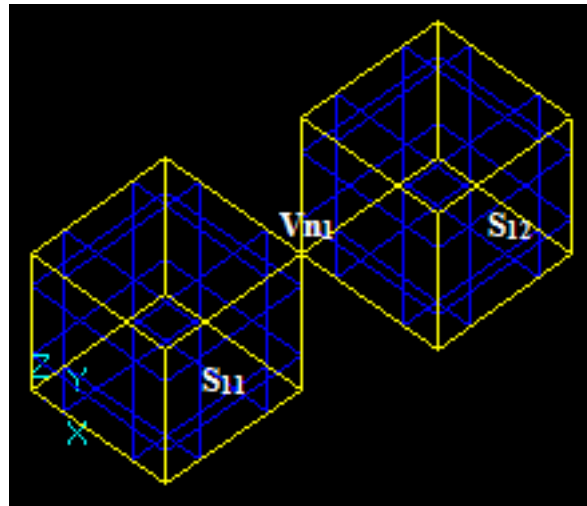


#### 8.4.22 Case 22: Two Solids that have a shared vertex.

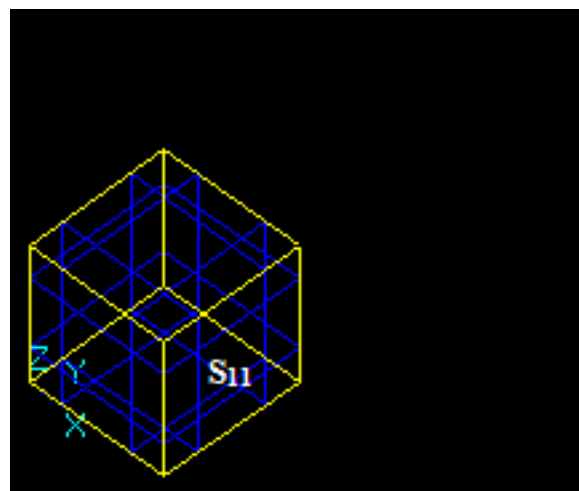
Let us consider two solids  $S1$  and  $S2$  that have a shared vertex:



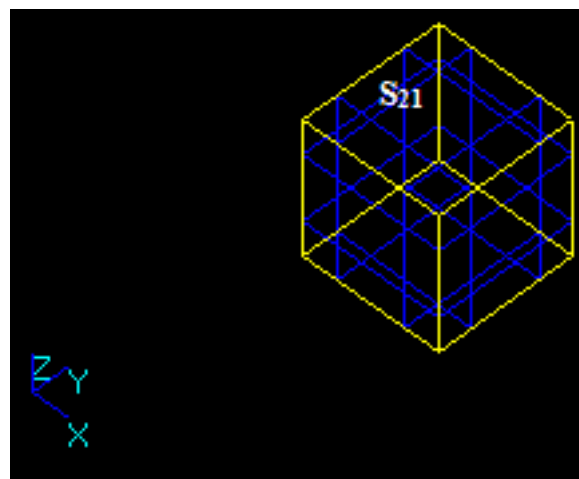
- The result of *Fuse* operation is a compound composed from the split parts of arguments i.e. 2 new solids  $S11$  and  $S21$ . These solids have one shared edge  $Vn1$ .



- The result of *Common* operation is an empty compound because the dimension (1) of the common part between *S1* and *S2* (vertex) is less than the lower dimension of the arguments (3).
- The result of *Cut12* operation is a compound containing split part of the argument *S1*. In this case argument *S1* has a common part with solid *S2* so the corresponding part is not included into the result.



- The result of *Cut21* operation is a compound containing split part of the argument *S2*. In this case argument *S2* has a common part with solid *S1* so the corresponding part is not included into the result.



### 8.5 Class BOPAlgo\_BOP

BOA is implemented in the class *BOPAlgo\_BOP*. The main fields of this class are described in the Table:

Name	Contents
<i>myOperation</i>	The type of the Boolean operation (Common, Fuse, Cut)
<i>myArgs[2]</i>	The arguments
<i>myDims[2]</i>	The values of the dimensions of the arguments
<i>myRC</i>	The draft result (shape)

The main steps of the *BOPAlgo\_BOP* are the same as of *BOPAlgo\_Builder* except for some aspects described in the next paragraphs.

## 8.6 Building Draft Result

The input data for this step is as follows:

- *BOPAlgo\_BOP* object after building result of type *Compound*;
- *Type* of the Boolean operation.

No	Contents	Implementation
1	For the Boolean operation <i>Fuse</i> add to <i>myRC</i> all images of arguments.	<i>BOPAlgo_BOP::BuildRC()</i>

| 2 | For the Boolean operation *Common* or *Cut* add to *myRC* all images of argument *S1* that are *Common* for the *Common* operation and are *Not Common* for the *Cut* operation | *BOPAlgo\_BOP::BuildRC()* |

## 8.7 Building the Result

The input data for this step is as follows:

- *BOPAlgo\_BOP* object the state after building draft result.

No	Contents	Implementation
1	For the Type of the Boolean operation <i>Common</i> , <i>Cut</i> and $myDim[0] < 3$	
1.1	Compute minimal dimension $Dmin = \min(myDim[0], myDim[1])$	<i>BOPAlgo_BOP::BuildShape()</i>
1.2	Make connexity blocks from sub-shapes of <i>myRS</i> , taking into account the value of <i>Dmin</i>	<i>BOPTools_Tools::MakeConnexity-Blocks()</i>
1.3	Build the result from shapes made from the connexity blocks	<i>BOPAlgo_BOP::BuildShape()</i>
2	For the Type of the Boolean operation <i>Common</i> , <i>Cut</i> and $myDim[0] = 3$	
2.1	$myShape = myRC$	<i>BOPAlgo_BOP::BuildShape ()</i>
3	For the Type of the Boolean operation <i>Fuse</i> and $myDim[0] = 3$	
3.1	Find internal faces ( <i>FWi</i> ) in <i>myRC</i>	<i>BOPAlgo_BOP::BuildSolid()</i>
3.2	Collect all faces of <i>myRC</i> except for internal faces ( <i>FWi</i> ) -> <i>SFS</i>	<i>BOPAlgo_BOP::BuildSolid ()</i>

3.3	Build solids ( <i>SDi</i> ) from <i>SFS</i> .	<i>BOPAlgo_BuilderSolid</i>
3.4	Add the solids ( <i>SDi</i> ) to the result	

## 9 Algorithm Limitations

The chapter describes the problems that are considered as Algorithm limitations. In most cases an Algorithm failure is caused by a combination of various factors, such as self-interfered arguments, inappropriate or ungrounded values of the argument tolerances, adverse mutual position of the arguments, tangency, etc.

A lot of bugs and failures of GFA algorithm can be caused by problems in low-level algorithms: Intersection Algorithm, Projection Algorithm, Approximation Algorithm, Classification Algorithm, etc.

- The Intersection, Projection and Approximation Algorithms are mostly used at the Intersection step. Their bugs directly cause wrong section results (i.e. incorrect section edges, section points, missing section edges or micro edges). It is not possible to obtain a correct final result of the GFA if a section result is wrong.
- The Projection Algorithm is used at the Intersection step. The purpose of Projection Algorithm is to compute 2D-curves on surfaces. Wrong results here lead to incorrect or missing faces in the final GFA result.
- The Classification Algorithm is used at the Building step. The bugs in the Classification Algorithm lead to errors in selecting shape parts (edges, faces, solids) and ultimately to a wrong final GFA result.

The description below illustrates some known GFA limitations. It does not enumerate exhaustively all problems that can arise in practice. Please, address cases of Algorithm failure to the OCCT Maintenance Service.

### 9.1 Arguments

#### 9.1.1 Common requirements

Each argument should be valid (in terms of *BRepCheck\_Analyzer*), or conversely, if the argument is considered as non-valid (in terms of *BRepCheck\_Analyzer*), it cannot be used as an argument of the algorithm.

The class *BRepCheck\_Analyzer* is used to check the overall validity of a shape. In OCCT a Shape (or its sub-shapes) is considered valid if it meets certain criteria. If the shape is found as invalid, it can be fixed by tools from *ShapeAnalysis*, *ShapeUpgrade* and *ShapeFix* packages.

However, it is important to note that class *BRepCheck\_Analyzer* is just a tool that can have its own problems; this means that due to a specific factor(s) this tool can sometimes provide a wrong result.

Let us consider the following example:

The Analyzer checks distances between couples of 3D check-points ( $P_i$ ,  $PS_i$ ) of edge  $E$  on face  $F$ . Point  $P_i$  is obtained from the 3D-curve (at the parameter  $t_i$ ) of the edge.  $PS_i$  is obtained from 2D-curve (at the parameter  $t_i$ ) of the edge on surface  $S$  of face  $F$ . To be valid the distance should be less than  $Tol(E)$  for all couples of check-points. The number of these check-points is a pre-defined value (e.g. 23).

Let us consider the case when edge  $E$  is recognized valid (in terms of *BRepCheck\_Analyzer*).

Further, after some operation, edge  $E$  is split into two edges  $E1$  and  $E2$ . Each split edge has the same 3D-curve and 2D-curve as the original edge  $E$ .

Let us check  $E1$  (or  $E2$ ). The Analyzer again checks the distances between the couples of check-points points ( $P_i$ ,  $PS_i$ ). The number of these check-points is the same constant value (23), but there is no guarantee that the distances will be less than  $Tol(E)$ , because the points chosen for  $E1$  are not the same as for  $E$ .

Thus, if  $E1$  is recognized by the Analyzer as non-valid, edge  $E$  should also be non-valid. However  $E$  has been recognized as valid. Thus the Analyzer gives a wrong result for  $E$ .

The fact that the argument is a valid shape (in terms of *BRepCheck\_Analyzer*) is a necessary but insufficient requirement to produce a valid result of the Algorithms.

#### 9.1.2 Pure self-interference

The argument should not be self-interfered, i.e. all sub-shapes of the argument that have geometrical coincidence through any topological entities (vertices, edges, faces) should share these entities.

**Example 1: Compound of two edges**

The compound of two edges  $E1$  and  $E2$  is a self-interfered shape and cannot be used as the argument of the Algorithms.



Figure 34: Compound of two edges

**Example 2: Self-interfered Edge**

The edge  $E$  is a self-interfered shape and cannot be used as an argument of the Algorithms.



Figure 35: Self-interfered Edge

**Example 3: Self-interfered Face**

The face  $F$  is a self-interfered shape and cannot be used as an argument of the Algorithms.



Figure 36: Self-interfered Face

**Example 4: Face of Revolution**

The face  $F$  has been obtained by revolution of edge  $E$  around line  $L$ .

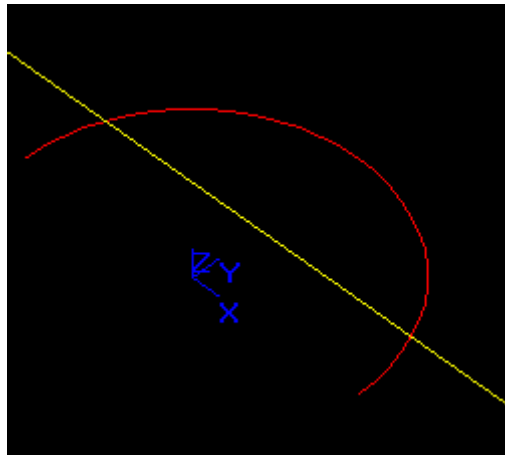


Figure 37: Face of Revolution: Arguments

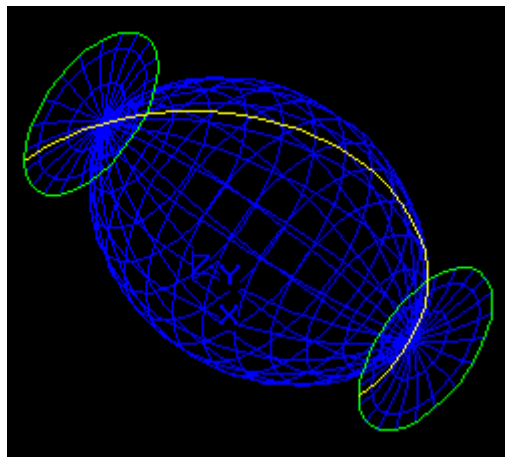


Figure 38: Face of Revolution: Result

In spite of the fact that face  $F$  is valid (in terms of *BRepCheck\_Analyzer*) it is a self-interfered shape and cannot be used as the argument of the Algorithms.

### 9.1.3 Self-interferences due to tolerances

#### Example 1: Non-closed Edge

Let us consider edge  $E$  based on a non-closed circle.

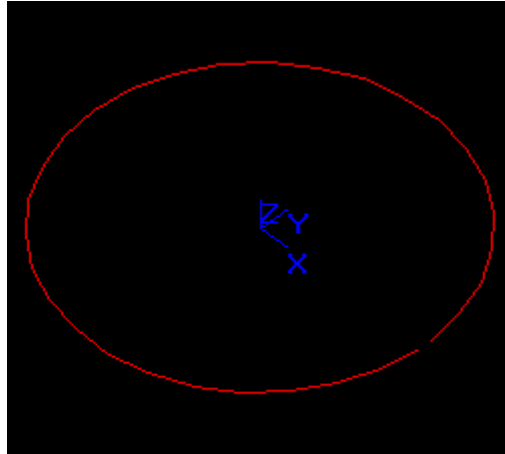


Figure 39: Edge based on a non-closed circle

The distance between the vertices of  $E$  is  $D=0.69799$ . The values of the tolerances  $Tol(V1)=Tol(V2)=0.5$ .

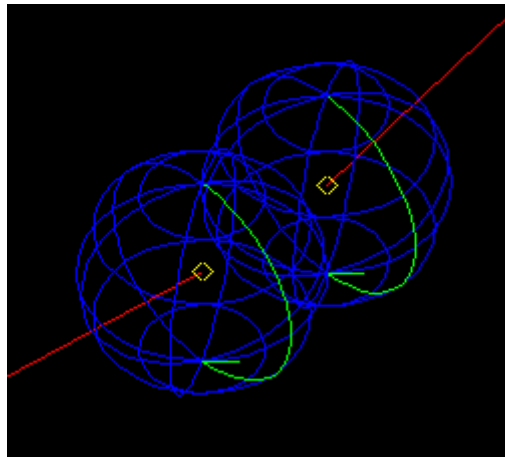


Figure 40: Distance and Tolerances

In spite of the fact that the edge  $E$  is valid in terms of *BRepCheck\_Analyzer*, it is a self-interfered shape because its vertices are interfered. Thus, edge  $E$  cannot be used as an argument of the Algorithms.

#### Example 2: Solid containing an interfered vertex

Let us consider solid  $S$  containing vertex  $V$ .

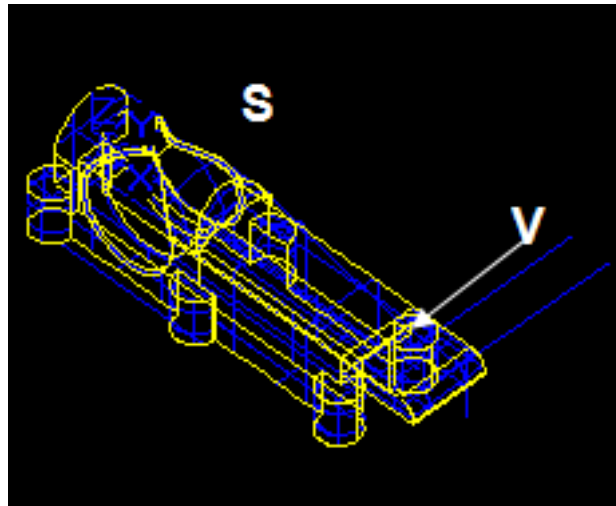


Figure 41: Solid containing an interfered vertex

The value of tolerance  $\text{Tol}(V) = 50.000075982061$ .

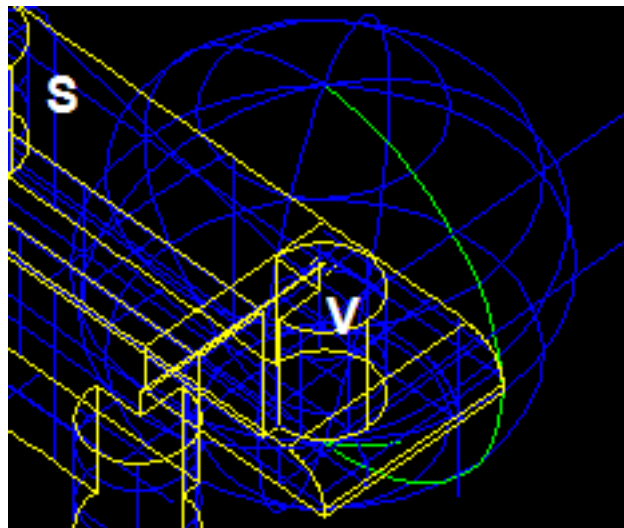


Figure 42: Tolerance

In spite of the fact that solid  $S$  is valid in terms of *BRepCheck\_Analyzer* it is a self-interfered shape because vertex  $V$  is interfered with a lot of sub-shapes from  $S$  without any topological connection with them. Thus solid  $S$  cannot be used as an argument of the Algorithms.

#### 9.1.4 Parametric representation

The parameterization of some surfaces (cylinder, cone, surface of revolution) can be the cause of limitation.

##### Example 1: Cylindrical surface

The parameterization range for cylindrical surface is:

$$U: [0, 2], \quad V: ]-, + [$$

The range of  $U$  coordinate is always restricted while the range of  $V$  coordinate is non-restricted.

Let us consider a cylinder-based *Face 1* with radii  $R=3$  and  $H=6$ .

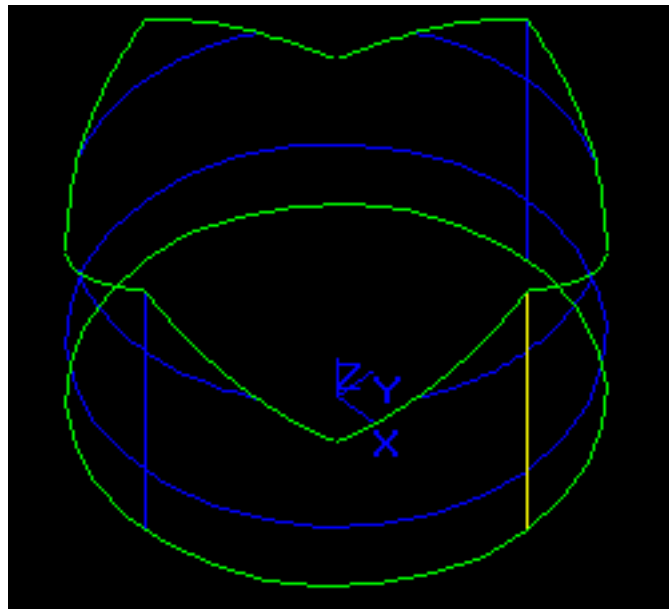


Figure 43: Face 1

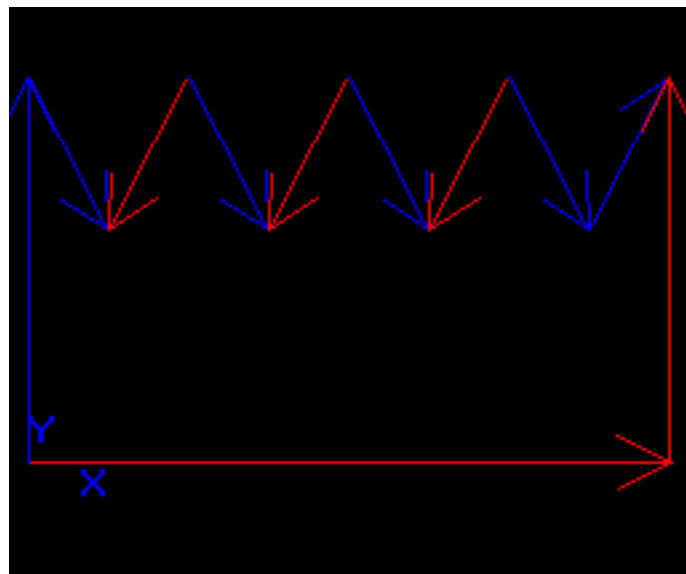


Figure 44: P-Curves for Face 1

Let us also consider a cylinder-based *Face 2* with radii  $R=3000$  and  $H=6000$  (resulting from scaling Face 1 with scale factor  $ScF=1000$ ).

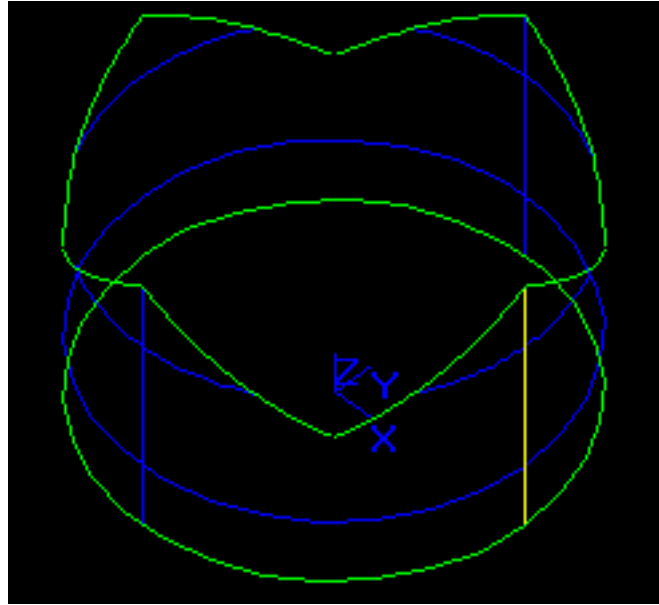


Figure 45: Face 2

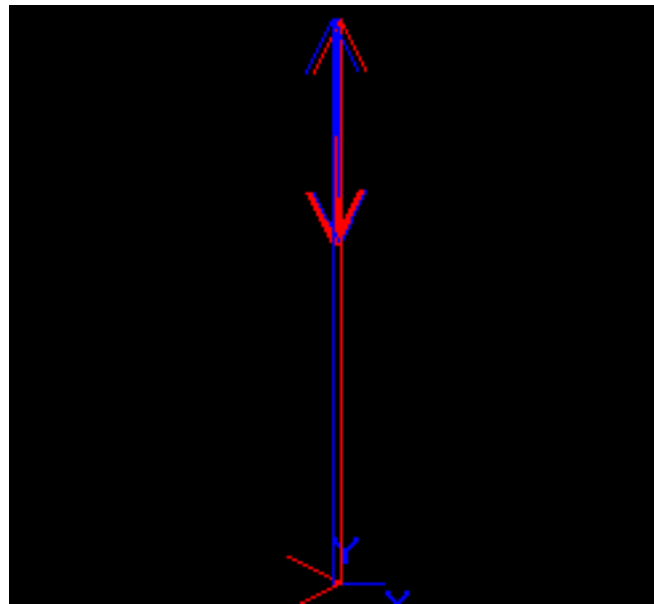


Figure 46: P-Curves for Face 2

Please, pay attention to the Zoom value of the Figures.

It is obvious that starting with some value of  $ScF$ , e.g.  $ScF > 1000000$ , all sloped p-Curves on *Face 2* will be almost vertical. At least, there will be no difference between the values of angles computed by standard C Run-Time Library functions, such as `double acos(double x)`. The loss of accuracy in computation of angles can cause failure of some BP sub-algorithms, such as building faces from a set of edges or building solids from a set of faces.

#### 9.1.5 How to fix gaps using tolerance

It is possible to create shapes that use sub-shapes of lower order to avoid gaps in the tolerance-based data model. Let us consider the following example:

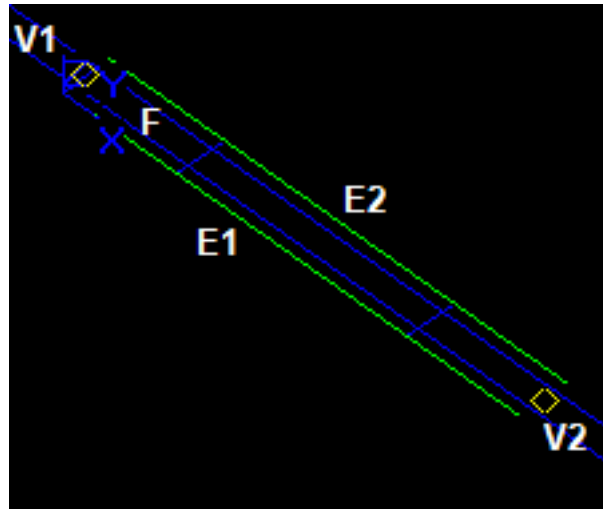


Figure 47: Example

- Face  $F$  has two edges  $E1$  and  $E2$  and two vertices, the base plane is  $\{0,0,0, 0,0,1\}$ ;
- Edge  $E1$  is based on line  $\{0,0,0, 1,0,0\}$ ,  $Tol(E1) = 1.e-7$ ;
- Edge  $E2$  is based on line  $\{0,1,0, 1,0,0\}$ ,  $Tol(E2) = 1.e-7$ ;
- Vertex  $V1$ , point  $\{0,0.5,0\}$ ,  $Tol(V1) = 1$ ;
- Vertex  $V2$ , point  $\{10,0.5,0\}$ ,  $Tol(V2) = 1$ ;
- Face  $F$  is valid (in terms of *BRepCheck\_Analyzer*).

The values of tolerances  $Tol(V1)$  and  $Tol(V2)$  are big enough to fix the gaps between the ends of the edges, but the vertices  $V1$  and  $V2$  do not contain any information about the trajectories connecting the corresponding ends of the edges. Thus, the trajectories are undefined. This will cause failure of some sub-algorithms of BP. For example, the sub-algorithms for building faces from a set of edges use the information about all edges connected in a vertex. The situation when a vertex has several pairs of edges such as above will not be solved in a right way.

## 9.2 Intersection problems

### 9.2.1 Pure intersections and common zones

#### Example: Intersecting Edges

Let us consider the intersection between two edges:

- $E1$  is based on a line:  $\{0,-10,0, 1,0,0\}$ ,  $Tol(E1)=2$ .
- $E2$  is based on a circle:  $\{0,0,0, 0,0,1\}$ ,  $R=10$ ,  $Tol(E2)=2$ .

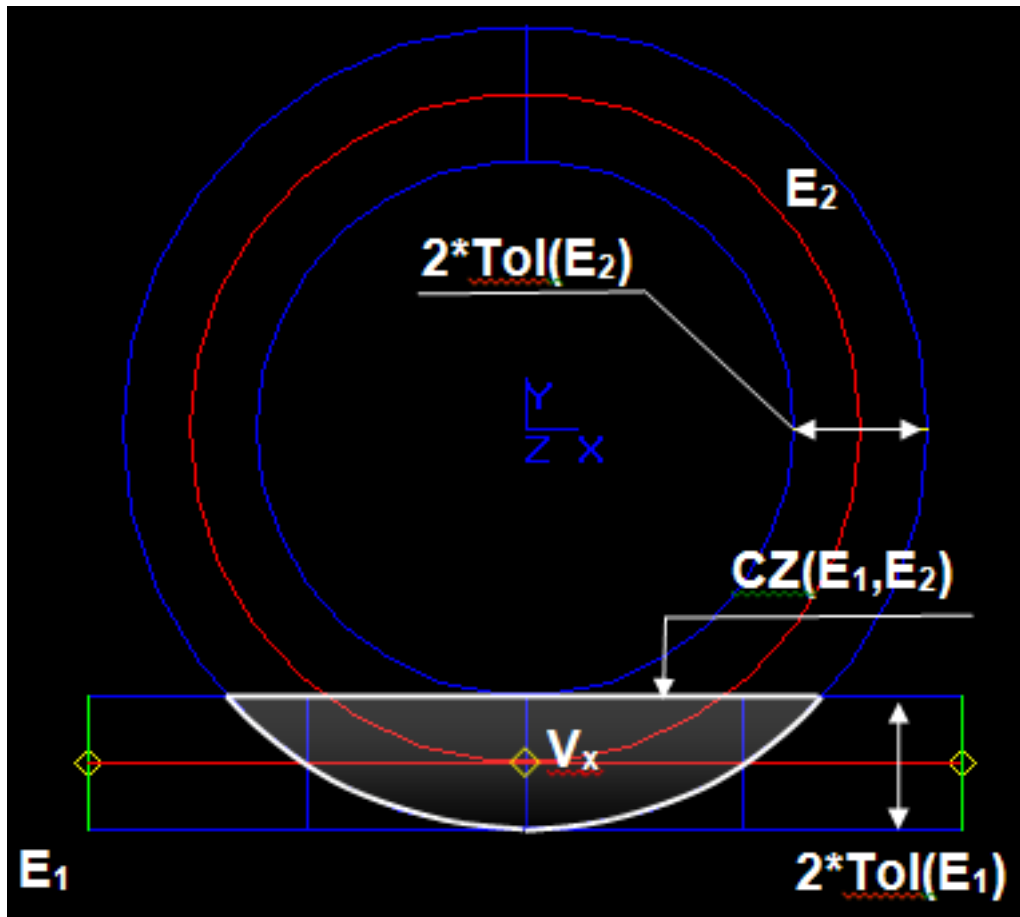


Figure 48: Intersecting Edges

The result of pure intersection between  $E1$  and  $E2$  is vertex  $Vx \{0, -10, 0\}$ .

The result of intersection taking into account tolerances is the common zone  $CZ$  (part of 3D-space where the distance between the curves is less than or equals to the sum of edge tolerances).

The Intersection Part of Algorithms uses the result of pure intersection  $Vx$  instead of  $CZ$  for the following reasons:

- The Algorithms do not produce Common Blocks between edges based on underlying curves of explicitly different type (e.g. Line / Circle). If the curves have different types, the rule of thumb is that the produced result is of type **vertex**. This rule does not work for non-analytic curves (Bezier, B-Spline) and their combinations with analytic curves.
- The algorithm of intersection between two surfaces *IntPatch\_Intersection* does not compute  $CZ$  of the intersection between curves and points. So even if  $CZ$  is computed by Edge/Edge intersection algorithm, its result cannot be treated by Face/Face intersection algorithm.

### 9.2.2 Tolerances and inaccuracies

The following limitations result from modeling errors or inaccuracies.

#### Example: Intersection of planar faces

Let us consider two planar rectangular faces  $F1$  and  $F2$ .

The intersection curve between the planes is curve  $C12$ . The curve produces a new intersection edge  $EC12$ . The edge goes through vertices  $V1$  and  $V2$  thanks to big tolerance values of vertices  $Tol(V1)$  and  $Tol(V2)$ . So, two straight edges  $*(E12, EC12)*$  go through two vertices, which is impossible in this case.

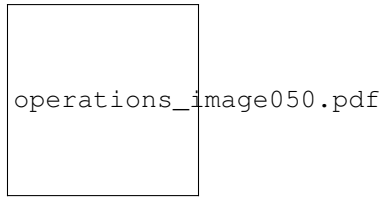


Figure 49: Intersecting Faces

The problem cannot be solved in general, because the length of  $E12$  can be infinite and the values of  $Tol(V1)$  and  $Tol(V2)$  theoretically can be infinite too.

In a particular case the problem can be solved in several ways:

- Reduce, if possible, the values of  $Tol(V1)$  and  $Tol(V2)$  (refinement of  $F1$ ).
- Analyze the value of  $Tol(EC12)$  and increase  $Tol(EC12)$  to get a common part between the edges  $EC12$  and  $E12$ . Then the common part will be rejected as there is an already existing edge  $E12$  for face  $F1$ .

It is easy to see that if  $C12$  is slightly above the tolerance spheres of  $V1$  and  $V2$  the problem does not appear.

#### Example: Intersection of two edges

Let us consider two edges  $E1$  and  $E2$ , which have common vertices  $V1$  and  $V2$ . The edges  $E1$  and  $E2$  have 3D-curves  $C1$  and  $C2$ .  $Tol(E1)=1.e^{-7}$ ,  $Tol(E2)=1.e^{-7}$ .

$C1$  practically coincides in 3D with  $C2$ . The value of deflection is  $Dmax$  (e.g.  $Dmax=1.e^{-6}$ ).

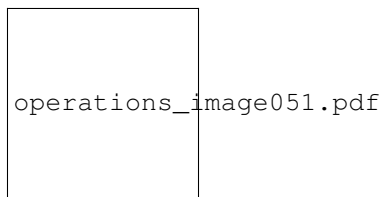


Figure 50: Intersecting Edges

The evident and prospective result should be the Common Block between  $E1$  and  $E2$ . However, the result of intersection differs.



Figure 51: Result of Intersection

However, the result contains three new vertices  $Vx1$ ,  $Vx2$  and  $Vx3$ , 8 new edges  $*(V1, Vx1, Vx2, Vx3, V2)*$  and no Common Blocks. This is correct due to the source data:  $Tol(E1)=1.e^{-7}$ ,  $Tol(E2)=1.e^{-7}$  and  $*Dmax=1.e^{-6}$ .

In this particular case the problem can be solved by several ways:

- Increase, if possible, the values  $Tol(E1)$  and  $Tol(E2)$  to get coincidence in 3D between  $E1$  and  $E2$  in terms of tolerance.

- Replace  $E1$  by a more accurate model.

The example can be extended from 1D (edges) to 2D (faces).

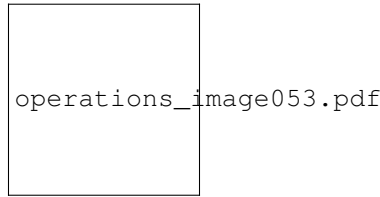


Figure 52: Intersecting Faces

The comments and recommendations are the same as for 1D case above.

### 9.2.3 Acquired Self-interferences

#### Example 1: Vertex and edge

Let us consider vertex  $V1$  and edge  $E2$ .

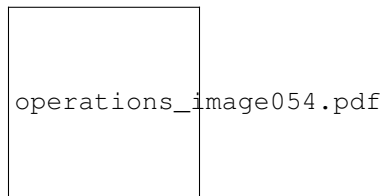


Figure 53: Vertex and Edge

Vertex  $V1$  interferes with vertices  $V12$  and  $V22$ . So vertex  $V21$  should interfere with vertex  $V22$ , which is impossible because vertices  $V21$  and  $V22$  are the vertices of edge  $E2$ , thus  $V21$  is not equal to  $V22$ .

The problem cannot be solved in general, because the length can be as small as possible to provide validity of  $E2$  (in the extreme case:  $Length(E2) = Tol(V21) + Tol(V22) + E$ , where  $e \rightarrow 0$ ).

In a particular case the problem can be solved by refinement of arguments, i.e. by decreasing the values of  $Tol(V21)$ ,  $Tol(V22)$  and  $Tol(V1)$ .

It is easy to see that if  $E2$  is slightly above than tolerance sphere of  $V1$  the problem does not appear at all.

#### Example 2: Vertex and wire

Let us consider vertex  $V2$  and wire consisting of edges  $E11$  and  $E12$ .



Figure 54: Vertex and Wire

The arguments themselves are not self-intersected. Vertex  $V2$  interferes with edges  $E11$  and  $E12$ . Thus, edge  $E11$  should interfere with edge  $E22$ , but it is impossible because edges  $E11$  and  $E12$  cannot interfere by the condition.

The cases when a non-self-interfered argument (or its sub-shapes) become interfered due to the intersections with other arguments (or their sub-shapes) are considered as limitations for the Algorithms.

## 10 Appendix

### 10.1 Building faces from set of edges

The algorithm builds a set of faces from a surface and a set of edges having p-curves on the surface.

#### 10.1.1 Terms and definitions

- **Original Face** - the source face  $F$  having an underlying surface  $S$  that is used as 2D domain.
- **ES** - a set of source edges having p-curves on surface  $S$ .
- **Contour** - a set of edges that is subset of  $ES$ .
- **Loop** - a closed Contour (in 2D).
- **Hole** - a Loop that has a negative mass property.
- **Growth** - a Loop that has a positive mass property.
- **Area** - a set of Loops within the one Growth and a number of Holes that are inside the Growth.
- **Deadlock** - a Contour that cannot be used as a Loop.
- **Result** - a set of faces LFR.

See the illustration for the terms in the image:

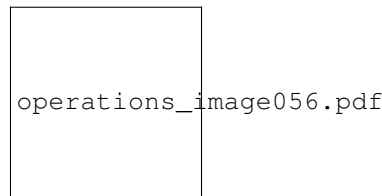


Figure 55: Terms and definitions

#### 10.1.2 Class BOPAlgo\_BuilderArea

The class *BOPAlgo\_BuilderArea* is a root class for implementations of algorithms to build areas (faces, solids) from a set of components (edges, faces).



Figure 56: Class inheritance diagram

The main fields of the class *BOPAlgo\_BuilderArea* are described in the Table:

Name	Contents
<i>myContext</i>	Pointer to the intersection Context
<i>myShapes</i>	Container for source shapes (edges, faces)
<i>myLoops</i>	Container for Loops
<i>myAreas</i>	Container for Areas
<i>myShapesToAvoid</i>	Container for Deadlocks

### 10.1.3 Class BOPAlgo\_BuilderFace

The class *BOPAlgo\_BuilderFace* implements the algorithm to build faces from a set of edges.

The main fields of the class *BOPAlgo\_BuilderFace* are described in the Table:

Name	Contents
<i>myFace</i>	Original face

The main steps of the algorithm are described in the Table:

No	Contents	Implementation
1	Collect the Deadlocks ( <i>myShapesToAvoid</i> ).	<i>BOPAlgo_BuilderFace::Perform-ShapesToAvoid()</i>
2	Build Loops ( <i>myLoops</i> ).	<i>BOPAlgo_BuilderFace::Perform-Loops()</i> , <i>BOPAlgo_WireSplitter</i>
3	Classify the Loops and build Areas ( <i>myAreas</i> )	<i>BOPAlgo_BuilderFace::Perform-Areas()</i>
4	Add internal shapes to the Areas	<i>BOPAlgo_BuilderFace::Perform-InternalShapes()</i>
5	Build the Result using the Areas and <i>myFace</i> .	<i>BOPAlgo_BuilderFace::Perform-InternalShapes()</i>

### 10.1.4 Class BOPAlgo\_WireSplitter

The class *BOPAlgo\_WireSplitter* implements the algorithm to build Loops from a set of edges *ES* and face *F*.

The main idea is to trace paths (in 2D) along the edges from the *ES* using the following conditions:

- Connectivity of the edges through vertices.
- Minimal clockwise angle (in 2D) between adjacent edges.
- Loop is closed.

See the illustration in the figure:



The input edges are *E1, E2...E16*. The edges in parentheses (*E11, E15*), (*E12, E16*), (*E9, E13*), (*E10, E14*) are the same with different orientations.

The angles *ij* are computed in the node *k* between the direction of the edge that enters in the node *Ei* and all directions of the edges *j* that leave the node.

- Let us start from edge *E3*;
- For node *A* and entering edge *E3* there are 3 leaving edges *E10, E15* and *E4* and three angles:

- $3,10 = \text{angle}(E3, E10)$ ,
- $3,15 = \text{angle}(E3, E15)$ ,
- $3,4 = \text{angle}(E3, E4)$ .
- The minimal clockwise angle is  $3,10$ , so:
  - edge  $E10$  will be next to edge  $E3$ ;
  - edge  $E2$  will be next to edge  $E10$ .
- The edges  $E3, E10$  and  $E2$  form Loop  $L1$ .
- Let us start from the next non-processed edge (e.g. for  $E4$ ).
- ... and so on

The main steps of the algorithm are as follows:

No	Contents	Implementation
1	Build connexity blocks $CBi(i=1,2 \dots NbCB)$ , where $NbCB$ is the number of connexity blocks from the ES.	<i>BOPAlgo_WireSplitter::MakeConnexityBlocks()</i>
2	For each connexity block $CBi$ : a) Compute angles $ij$ for each node (for entering and leaving edges); b) Compute Loops	<i>BOPAlgo_WireSplitter::SplitBlock()</i>

## 10.2 Building solids from set of faces

The algorithm is to build a set of solids from a set of faces.

### 10.2.1 Terms and definitions

- **FS** - a set of source faces.
- **Contour** - a set of faces that is subset of FS.
- **Loop** - a closed Contour (in 3D).
- **Hole** - a Loop that has a negative mass property.
- **Growth** - a Loop that has a positive mass property.
- **Area** - a set of Loops within the one Growth and a number of Holes that are inside the Growth.
- **Deadlock** - a Contour that cannot be used as a Loop.
- **Result** - a set of solids  $LSo$ .

### 10.2.2 Class BOPAlgo\_BuilderSolid

The class *BOPAlgo\_BuilderSolid* contains the implementation of the algorithm to build solids from a set of faces. The content of the main steps of the algorithm is described in the Table.

No	Contents	Implementation
1	Collect the Deadlocks ( <i>myShapesToAvoid</i> ).	<i>BOPAlgo_BuilderSolid::PerformShapesToAvoid()</i>

2	Build Loops ( <i>myLoops</i> ).	<i>BOPAlgo_BuilderSolid::Perform-Loops()</i>
3	Classify the Loops and build Areas ( <i>myAreas</i> ).	<i>BOPAlgo_BuilderSolid::Perform-Areas()</i>
4	Add to Areas the internal shapes.	<i>BOPAlgo_BuilderSolid::Perform-InternalShapes()</i>
5	Build the Result using the Areas and <i>myFace</i> .	<i>BOPAlgo_BuilderSolid::Perform-InternalShapes()</i>

### 10.3 Packaging

The following packages contain the implementation of the algorithm:

No	Name	Contents
1	<i>BOPCol</i>	Collection classes for all algorithms
2	<i>BOPInt</i>	Auxiliary classes for IP
3	<i>BOPDS</i>	DS and a set of auxiliary classes for DS
4	<i>BOPTools</i>	Auxiliary classes for BP
5	<i>BOPAlgo</i>	IP, BP
6	<i>BOPTest</i>	Testing commands for Draw application