GIMA

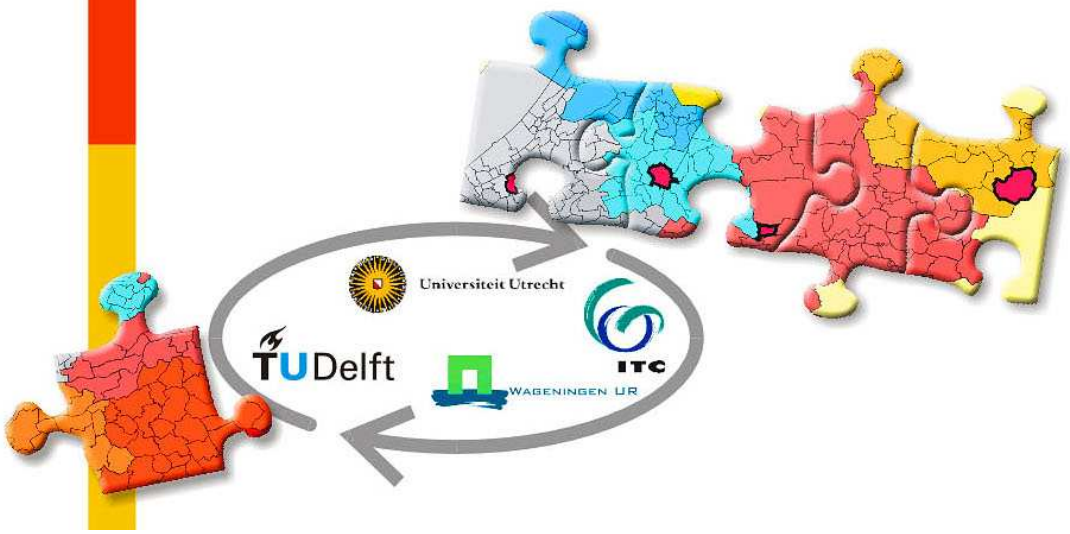
Geographical Information Management and Applications

Thesis v1.0
GIMA MSc

3D topological structure management within a DBMS

validating a topological volume

Bregje Brugman
04 June 2010
bregjebrugman@gmail.com



Professor:
prof. dr. ir. P.J.M. van Oosterom (TUDelft)

Supervisor:
drs. T.P.M. Tijssen (TUDelft)

Reviewer:
dr.ir. R.J.A. van Lammeren (Wageningen UR)

Table of contents

Summary 5
Terms 7
1. Introduction 9
1.1. Data management in a geo-DBMS 9
1.2. Problem definition 10
1.3. Research goals, objectives and questions 11
1.4. Research methodology 12
1.5. Report structure 14
2. (3D) Topological structures 15
2.1. The state of art of topological structures 15
2.2. Existing topological structures 16
2.2.1. Winged edge (Dutch Cadastre) 17
2.2.2. Oracle's topological structure 17
2.2.3. Radius Topology (1Spatial) 18
2.2.4. 3D FDS 19
2.2.5. SSS 20
2.2.6. TEN 20
2.3. Conclusion 20
3. Design of a 3D topological structure 23
3.1. The requirements for a 3D topological structure 23
3.1.1. Orientation 23
3.1.2. Boundaries 24
3.1.3. Singularities 27
3.1.4. Universal volume 28
3.1.5. Geometrical realization 28
3.2. The conceptual model 30
4. Validation of the 3D topological structure 33
4.1. Introduction 33
4.2. Existing validation functions 34
4.3. A valid 3D topological structure 37
5. Implementation of the 3D topological structure (prototype) 41
5.1. Topological structure (logical model) 41
5.2. The implementation (technical model) 44
5.3. Validation tests 45
5.3.1. Test 1) unique primitives 47
5.3.2. Test 2) primitive references 48
5.3.3. Test 3) each face/volume consists of one outer boundary 49
5.3.4. Test 4) closed boundaries 49
5.3.5. Test 5) proper orientation 50
5.3.6. Test 6) linear structure 50
5.3.7. Test 7) no intersections 51
5.3.8. Test 8) bounding single volumes/areas 51
5.4. Illustrations of validation 52
5.5. Geometry operations 55
6. Test and evaluation 59
6.1. Testing 59
6.2. Evaluation 65
Conclusion 69
References 71
Appendix I – PL/SQL-scripts 73

Summary

The goal of this research is to develop a 3D topological structure with validation functionality and a conversion function.
Based on this research question, some sub questions have been defined and a research methodology. The outline of this research is elaborated in chapter 1.

The first use of topology has been attributed to Euler in 1736, since then, topology has evolved in mathematics but also in GIS. Since the second half of the 20th century, 2D topological data structures are historically well established, with structures like TIGER and GBF/DIME. Several 3D topological structures have been developed as well. Most of them by researchers (for example 3D FDS and SSS). No commercial geo-DBMS has implemented a 3D topological structure, until recently. 1Spatial has developed a 3D topological structure. This structure has been analyzed among other 2D and 3D structures (chapter 2).

It is clear that the existing structures differ a lot. Some structures maintain a 3D primitive while others do not. Orientation is stored in different ways, the same applies for geometry and singularities. Some structures have quite some redundancy, while other structures have only a few relationships stored explicitly, which makes maintaining the structure harder. In spite of the many differences, the characteristics of, both 2D and 3D, topological structures are based on the same aspects: dimension, partition (including the universe), primitives (including their relationships), orientation, singularities and geometrical realization.

Based on these main aspects, the requirements for the topological structure are defined in chapter 3 and a conceptual model is designed. The model consist of four primitives, which are related to each other by their (co)boundary relationships. These boundary relationships exact a full space partition, where every primitive involved, is part of a volume primitive. No isolated and dangling primitives are allowed. Next to the boundaries, especially the rings and the shells, orientation plays an important role within a topological structure and the geometrical realization. Single geometries have their advantages, therefore single geometries and topological structures should be used together.

Once the structure has been defined, validation rules need to be set. In order to validate a 3D topological structure, the involved volumes must be valid as well as the whole structure, which means the relationships between the volumes. Since no single definition of a valid 3D primitive is available, the rules are based on a few existing validation functions for 3D single geometries. No 3D topological validation function exists at the moment.
Once the rules for a valid structure are set (chapter 4), the structure has been implemented into Oracle Spatial. The validation rules are translated into validation tests and implemented on this structure (chapter 5), as well as the geometry operations.

In chapter 6, the structure, validation tests and geometry operations are tested with a test data set and evaluated.

Terms

In order for a clear and consistent use of terms, the terms listed below will be used in this report (terms from used sources will be translated into these terms).

Topological primitives	Geometrical primitives
Node	Point
Edge	Line/linestring
Face	Polygon/surface
Volume	Solid

2D	3D
2D polygon	polygon
2D surface	Surface
Hole	Void
Boundary	Bounding surface
Ring	Shell
Inner – outer	Inner - outer
Interior – exterior	Interior - exterior

1. Introduction

3D spatial data management is becoming more and more important at the moment. For sectors like urban planning, emergency services, hydrology and telecommunication, 3D data management would be very helpful. Furthermore the availability of 3D data is growing due to new data acquisition techniques. Traditionally a geo-DBMS is the best software to deal with spatial data management, it is able to store and manage large amounts of data. Geo-DBMSs are well developed for 2D spatial data management, but underdeveloped for the third dimension. Ideally a 3D GIS would have the same functions as a 2D GIS. This would 'simply' mean an expansion from second dimension to the third dimension, where new spatial data structures and spatial data relationships need to be defined (Abdul Rahman, Zlatanova & Pilouk 2001). For the past 30 years a lot of research has been carried out and a lot of progress has been made in the field of 3D spatial data management. GEO++ is an early example of a 3D GIS, based on a geo-DBMS (van Oosterom, Vertegaal & van Hekken, 1994).

In this chapter the research will be defined. In the first section (section 1.1) an introduction on data management in geo-DBMSs will be discussed. After defining the problem definition in section 1.2, the research goals, objectives and questions will be set (section 1.3). Then the research methodology will be explained, followed by the structure of this report.

1.1. Data management in a geo-DBMS

Nowadays a DBMS (Database Management System) is a good way for storing spatial data. Traditionally databases store administrative data, but they have developed their 'geo' component and are able to store spatial data. Most mainstream DBMSs currently support spatial data types and spatial functions built on these spatial data types. This so-called integrated architecture, where geometric data is stored together with administrative data in one DBMS, is used more and more for spatial data management purposes and is often called a geo-DBMS. The main advantages of using an integrated architecture are the capability of a DBMS to handle large volumes of data, the ability to ensure the logical consistency and integrity of data and the ability of multi-user control. Most spatial data types within a DBMS are defined as a single geometry, which describes a geometry within a coordinate reference system. For some purposes, like finding neighbours or managing data in structured in a partition, a topological structure will be more suitable. A topological structure describes the relationships between the primitives (node, edge, face and/or volume) of an object.
The basic 2D single geometries and 2D functions are standardized by the Open Geospatial Consortium (OGC) and the International Organization for Standardization (ISO) in the Simple Feature Specification (SFS) (Ryden 2005; ISO/TC211 2004). Many mainstream geo-DBMSs like Oracle Spatial (Oracle), Informix Spatial DataBlade (IBM), DB2 Spatial Extender (IBM), PostGIS (PostgreSQL) and Ingres now support 2D single geometries (most of them according to the standards). Also many (2D) spatial functions (on single geometries) are implemented, but 'real' 3D functions are hardly implemented. The 3D functions which are available are usually based on 2D objects that appear in 3D space (Khuan, Abdul Rahman & Zlatanova 2008). Most geo-DBMSs have included 3D coordinates within their single geometries. This means usually that each x,y coordinate has (only) one z-value. This is often referred to as 2,5D. This option of storing 3D coordinates (x,y,z) within the geo-DBMS makes it possible to model 3D with 2D primitives. A 3D object could be modelled, for example, by combining several polygons. Modelling in this way is very much restricted, some spatial functions do not work, for example the validation of the object as a whole. It is mainly good for visualization. In case of 'real' 3D, only some 3D single geometries and 3D spatial functions are developed within geo-DBMSs, for example Oracle Spatial (11g), which has a 3D solid implemented

and a few 3D functions.[1] 3D topological structures are even less developed than 3D single geometries. Very few 3D topological structures have been developed, usually by researchers. Only Radius Topology has recently made the first steps towards the implementation of a 3D topological structure.

1.2. Problem definition

As mentioned before data types can be defined in two ways, based on a single independent geometry or a topological structure.[2] A single geometry describes a geometry within a coordinate reference system, by using the (geometric) primitives point, line, polygon and/or solid (Khuan, Abdul Rahman & Zlatanova 2007). This means that the entire geometry is stored in one geometry column of a table, by an ordered sequence of coordinates stored as x,y pairs (2D) and possibly extended by the Z-ordinate. Oracle has recently (since Oracle 11g) implemented a (3D) solid (SDO_GEOMETRY) (Murray et al. 2008a). Also researchers have designed 3D solids (single geometries) and implemented prototypes with corresponding functions within geo-DBMSs, two examples are GEO++ of van Oosterom, Vertegaal & van Hekken (1994) and the polyhedron of Arens, Stoter & van Oosterom (2005). A more extensive overview is given by Khuan, Abdul Rahman & Zlatanova (2007). Next to linear 3D solids, i.e. the polyhedron, also non-linear data types (like freeform curves and surfaces, CAD-objects and cones and spheres) have been developed, but those will be left out of scope of this research (Zlatanova 2008).

A topological structure describes the relationships between the primitives (node, edge, face and/or volume). The main property of topology is the invariance under topological transformations, i.e. rotation, scaling and translation. This means the topological relationships remain unchanged under any transformation, which makes a topological structure an appropriate approach for maintaining spatial relationships within a DBMS and performing relationship functions (Zlatanova, Abdul Rahman & Shi 2004). Single geometries (2D) have been supported for a longer time, but also topological structures are currently more and more supported within DBMSs. Oracle for example has implemented a (2D) topological structure (SDO_TOPO_GEOMETRY) (Murray et.al. 2008b). 1Spatial has recently developed a 3D topological structure. Although spatial analysis can be performed on both single geometries and topological structures, topological structures are important for performing certain functions, like finding neighbours and maintaining data structured in a partition. In general one can say the functions related to geometry can best be performed on the single geometry and functions related to the spatial relationships on the topological structure. Two kinds of functions are particularly suitable for topological structures: the relationship functions (like neighbour functions) and the network functions (network topology is out of scope of this thesis) (Zlatanova, Abdul Rahman & Shi 2004). Therefore not one definition (geometrical or topological) will be best suitable for all functions. For different purposes different structures will be optimal and a function to derive one from the other will be necessary. Therefore both structures are often used in one application (and a conversion function is designed). In table 1.1 an overview of the main advantages of both approaches is presented (Van Oosterom, Stoter, Quak & Zlatanova 2002).

[1] Oracle Spatial (11g) has four 'real' 3D functions that make use of a 3D spatial index: SDO_ANYINTERACT, SDO_FILTER, SDO_NN and SDO_WITHIN_DISTANCE.
[2] Complex geometries could have internal topologies.

Topological structure	Single geometry
• Less redundant storage • Easy to maintain consistency of data • Efficient queries for relationship functions	• Easy to implement • Geometries modelled in a single column • Efficient queries for metric functions

Table 1.1 Advantages of topological structures versus single geometries

Before performing any spatial analysis on a topological structure or single geometry, the structure including its primitives need to be valid (in 3D this includes the volume or solid primitive). When data is added or updated the topological rules need to be validated. Therefore a validation function is needed. 2D validation functions exist mainly for single geometries, but also for topological structures (like Oracle's 2D topological validation function). 3D validation functions also exist but are rather rare. No real 3D validation functions for topological structure exists. 1Spatial has a 3D topological structure developed and is using a validation function based on Oracle's 2D topological validation function and Oracle's 3D single geometry validation function.[3] For two reasons 1Spatial's validation function is not a complete 3D topological validation function. First, the validation rules apply to single primitives and not the whole structure. Second, for validation 1Spatial has pointed out a minimum set of validation rules, these rules are not enough to guarantee a valid primitive (e.g. volumes are not checked for a contiguous volume or a proper orientation).[4]

For single 2D geometries, standards for valid polygons are set in the Simple Feature Specification (of the OGC and ISO) (Ryden 2005; ISO/TC211 2004). The Complex Feature Specification for topology is not finished yet, in terms of implementation specifications. Van Oosterom, Quak & Tijssen (2004) pointed out that even the standards for valid (geometrical) polygons are not unambiguous and complete. Also the interpretations of different DBMS vendors differ from each other and from the standards. Implementation specifications for 3D primitives (both geometrical and topological) are not set yet.

1.3. Research goals, objectives and questions

This research will focus on 3D topological structure management within a geo-DBMS, whereas the most important aspects are data management of a space partition, and validation. The combination of a 3D topological structure with a 'real' 3D topological validation function will be, at this time as far as the author knows, new.
The topological structure will be an efficient structure with less redundancy, which will be implemented within Oracle Spatial including validation and conversion functionality. Oracle Spatial is chosen because it is a widely used DBMS with a 3D single geometry primitive implemented. As pointed out, the interaction between single geometries and topological structures is very important. The topological structure will make use of a polyhedral primitive (and not, for example, of a TEN). The ISO standard 19107 will serve as input. It provides a conceptual spatial schema. An overview of the research goals with their accompanying objectives is presented in table 1.2.

[3] No implementation specifications are available.
[4] More about validation rules and validation functions in chapter 4.

Goals	Objectives
Develop a 3D topological structure within a geo-DBMS	• Design a 3D topological structure (which supports a validation and conversion function) • Implement a prototype of the developed 3D topological structure within Oracle Spatial
Develop a validation function on the developed 3D topological structure within the geo-DBMS	• Define validation rules • Design a validation function • Implement a prototype of the designed validation function within Oracle Spatial on the developed 3D topological structure
Develop a conversion function on the developed 3D topological structure within the geo-DBMS	• Design a conversion function • Implement a prototype of the conversion function within Oracle Spatial on the developed 3D topological structure

Table 1.2 Research goals and objectives

Based on the goals and objectives mentioned above, the main research question and some sub questions are formulated.

Main research question:

How to design and develop a 3D topological DBMS structure which supports validation and a conversion function?

Sub questions:

What are the requirements for a 3D topological structure?

- What are the advantages and disadvantages of existing 2D and 3D topological structures?
- How to extend a 2D topological structure to a 3D topological structure?

How to define the best suitable 3D topological structure?

- Which primitives will be used in the topological structure?
- Which references will be stored?

How to define the best suitable validation function?

- What are the advantages and disadvantages of existing 2D and 3D validation functions?
- What is the definition of a valid 3D topological structure?
- How to validate a 3D topological structure?

How to define the best suitable conversion function from a topological primitive to a geometrical primitive?

- What are the output requirements?

In the next section the methodology is explained on how this research will be carried out to answer the above questions and get to the goals and objectives of this research.

1.4. Research methodology

To clarify the scope of this research, some related items which will not be in this research are listed below (table 1.3) as well as some assumptions. Most important aspects which are not in this research are dealing with tolerances, writing efficient algorithms, updating the structure and feature modelling.

Out of scope	• Capability of handling large data volumes • Acquiring 3D data • Temporal modelling • 3D visualization • Data integration (for example from different front-end applications) • Other data types (like non-linear volumes) • Implementation in a middleware or front-end application • Cleaning test data • Develop other functions (than conversion and validation functions) • Editing/updating structure • Dealing with tolerance values • Feature modelling • Efficient algorithms, which leads to efficiency in time
Assumptions	• A linear topological structure will be designed • Structure and functions will be ISO 19107 compliant • Implementation within Oracle Spatial (11g) • Test data is clean (tolerance value = 0)

Table 1.3 Out of scope and assumptions

The research will be carried out according to a generic GIS development methodology consisting of four main phases between the initiation and close-out: analysis, design, implementation and evaluation (Reeve & Petch 1999). Analysis will be applied to 2D and 3D topological structures and validation functions. Design, implementation and evaluation will be applied to a 3D topological structure, validation function and conversion function.

To get to a final 3D topological structure, 2D topological structures will be investigated first. Then in line with 2D structures, a 3D structure will be designed. To get a suitable validation function, a valid volume/3D structure will be defined first. Next the topological structure will be implemented and with a conversion function and validation function. Finally some testing will be done on the structure and the developed functions with a test data set. The final deliverable will consist of a report and a prototype of a 3D topological structure with a validation and conversion function.

In the first part of the research, the analysis phase, studying will be done based on literature of existing structures and functions as well as studying current DBMS structures and functions (see references). For developing purposes Oracle Spatial will be used and the algorithms will be written in PL/SQL. This language, developed by Oracle, is using SQL to access the data and is easy to implement. For visualization purposes (only) Bentley MicroStation will be used. There will be no focus on visualization in this research, it will only be used to check results during the research and illustrate the implementations. An overview of the research materials is given in table 1.4.

Material	**Details**
Literature	See references
Hardware	Dell Inspiron I6000 Intel® Pentium® M Processor 1.60 GHz Microsoft XP – Home edition, version 2002 with service pack 3
Data	The TUDelft campus data set (3D)
Software	Oracle (Spatial) 11g (release 2) Bentley (MicroStation), Bentley Map V8*i* (version 08.11.05.26) Microsoft Office 2003 (Word, PowerPoint and Excel)

Table 1.4 Details of research materials

1.5. Report structure

This report is structured according to the main research phases (analysis, design, implementation and evaluation). First some background on topological structures will be presented in chapter 2, including some examples of several existing 2D and 3D topological structures. Then the design of a topological structure will be discussed, including a conceptual model. Chapter 4 deals with validation. The implementation of the designed structure and the validation and conversion functions within Oracle Spatial will be presented in chapter 5 and in chapter 6 the structure and the functions will be tested and evaluated. The close up will be the overall conclusion.

2. (3D) Topological structures

The first use of topology as a tool has been attributed to Euler, who proved that the Königsberg Bridge Problem is insoluble in 1736 (in a paper entitled 'The solution of a problem relating to the geometry of position')[5]. The problem asks if the seven bridges of the city of Königsberg (since 1946 known as Kaliningrad, Russia), over the river Preger can all be traversed in a single trip without doubling back, with the additional requirement that the trip ends in the same place it began. This problem was answered negative by Euler and represented the beginning of the graph theory (introducing the Eulerian cycle: a graph cycle (starts and ends at the same graph vertex) which uses each graph edge exactly once). In 1750 Euler gave his formula for a planar partition 'v-e+f=2' (where v is the number of vertices, e the number of edges and f the number of faces). This was the first time a polyhedron was not approached in a purely geometric way and formed the basis for topology, although the formula applies only to convex polyhedra without any voids (O'Connor & Robertson 1996). Since then, topology has evolved as a branch of mathematics. Since the second half of the 20th century 2D topological data structures are historically well established.

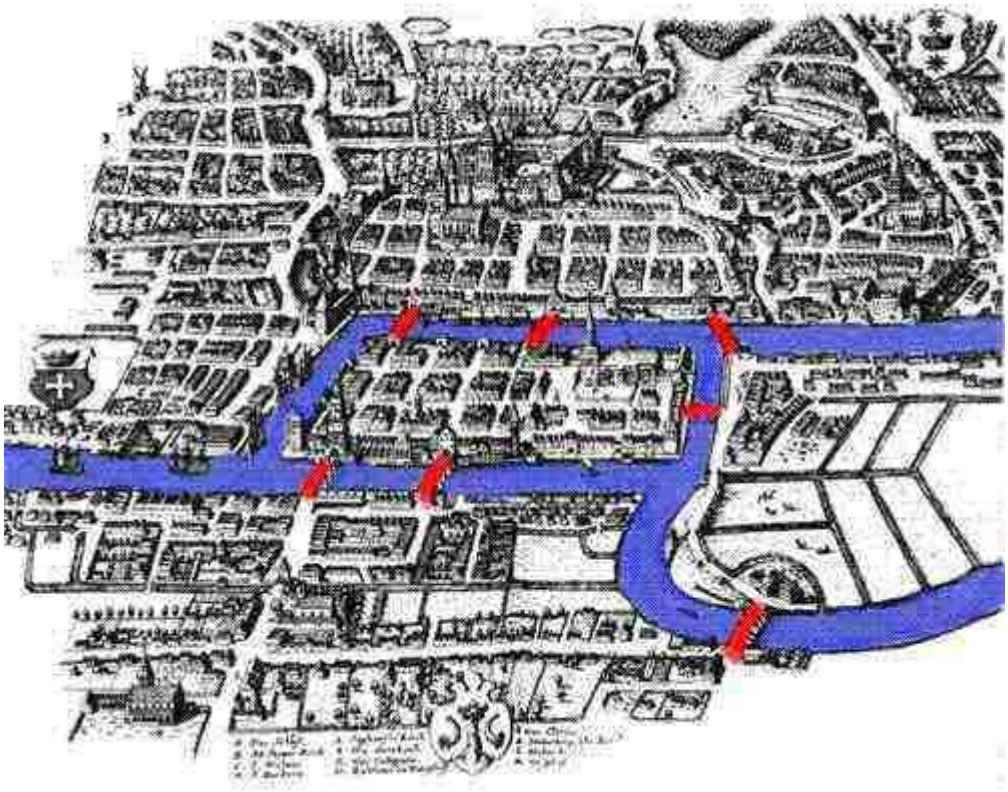


Figure 2.1 Königsberg and its seven bridges (O'Connor & Robertson 1996)

In this chapter, topological structures will be analyzed. Starting with a some background on topological structures in section 2.1. In section 2.2 several existing 2D and 3D topological structures will be discussed, after which, a conclusion on the discussed structures will follow.

2.1. The state of art of topological structures

Many GIS packages have constructed 2D topological structures (e.g. ESRI's coverages, Geomedia Professional (Intergraph) and Smallworld (GE Energy)), some CAD packages provide tools to check topological consistency (e.g. MicroStation) and some DBMSs have 2D topological implementations (e.g. Oracle Spatial) (Zlatanova, Abdul Rahman & Shi

[5] Originally: 'Solutio problematis ad geometriam situs pertinentis'.

2004). Radius Topology (1Spatial) is an extension for Oracle Spatial and recently developed a 3D implementation (www.1spatial.com/: consulted May 2010). Most of them support both network (cables, roads etc.) and spatial partition (e.g. a cadastral map) topology management, however this research will focus on spatial partition. 2D topological structures are quite well developed in contrary to 3D structures, which are still being researched and are poorly developed at the moment. None of the commercial products currently offer 3D topological functionality.

2D topological data structures are historically well established, e.g. POLYVRT (Peucker and Chrisman, 1975), TIGER (Topologically Integrated Geographic Encoding and Referencing system, US Census Bureau) and GBF/DIME (Geographic Base File/Dual Independent Map Encoding, US Census Bureau, 1969) (Ellul, Haklay & Bevan 2005; Boudriault 1987). Most of the 2D topological structures are based on the planar graph theory. A planar graph can be represented on a plane with nodes at each intersection between edges. Each given oriented edge has one face on the right, and one face on the left (De la Losa & Cervelle 1999).
A 3D topological structure can be supported by the *boundary representation* approach, like 3D FDS (Molenaar 1990), which utilises a series of topological primitives (node, edge, face and volume) for representing a (3D) volume (Ellul, Haklay & Bevan 2005; Ellul 2007). The interior of the volume is represented by the space enclosed by the bounding surface. The bounding surface consists of faces which are bounded by edges. Nodes define the edges.
Other main approaches are the constructive solid geometry approach and the simplicial complexes approach. In the *constructive solid geometry (CSG)* approach, where Boolean operators are used on simple primitives to model complex objects, topology is not represented explicitly (Groger & Plumer 2005). This approach is therefore not suitable for this research. The *simplicial complexes* approach divides an object into a number of tetrahedrons (each consisting of exactly four faces). Examples are the TEN structure of Pilouk and Penninga (2008). Between the boundary representation and simplicial complex approaches, the *regular polytope* approach can be distinguished. This approach uses convex regular polytopes, which are defined as the intersection of a finite set of half spaces (Thompson & van Oosterom 2009).

In order to develop a successful 3D topological structure, a thorough understanding of 2D structures is necessary. In the next section some examples of existing 2D and 3D structures will be presented.

2.2. Existing topological structures

This section will present some examples of existing 2D and 3D topological structures supporting planar/space partition.

Van Oosterom, Stoter, Quak & Zlatanova (2002) and Zlatanova, Abdul Rahman & Shi (2004) present overviews of different topological structures. The characteristics of the different structures are based on several main aspects: space/plane partition, used primitives, constructive rules, orientation and explicit/implicit relationships. The most suitable 3D topological structure depends on the type of application it is used for. There is no single 3D topological structure suitable for all types of applications. For this reason Van Oosterom, Stoter, Quak & Zlatanova propose the offering of multiple topological structures in one database by describing the objects, rules and constraints of each structure in a metadata table (Van Oosterom, Stoter, Quak & Zlatanova 2002).
Next several existing 2D and 3D topological structures will be presented in more detail:[6]

- Winged edge (widely used in 2D topology); the Dutch Cadastre has been highlighted as example.
- The (2D) topological model of Oracle (also based on the winged edge method) SDO_TOPO_GEOMETRY with a reference to a similar model implemented in PostGIS.

[6] Feature modelling is out of scope, therefore the features in the topological structures will not be discussed.

- Radius Topology which is implemented in Oracle as well, but with a different approach, both 2D and 3D.
- 3D FDS (Formal Vector Data Structure) as example of a 3D boundary representation approach. 3D FDS has been the first data structure to consider spatial objects as an integration of geometric and thematic properties.
- SSS (Simplified Spatial Schema) as an example of a 3D structure which uses only two primitives.
- TEN (TEtrahedral irregular Network) as a simplicial based approach for a comparison with the boundary representation approach.

2.2.1. Winged edge (Dutch Cadastre)

Many 2D topological structures are based on winged edge structure of Baumgart (1975), in which the edges play the crucial role. A polygon is described by an ordered set of edges (counter clockwise for the exterior). Each edge has an orientation by defining its start- and endnode. Furthermore each edge has a left and right face defined and four 'wings'. The wings are the next and previous right and left edges (see figure 2.2).

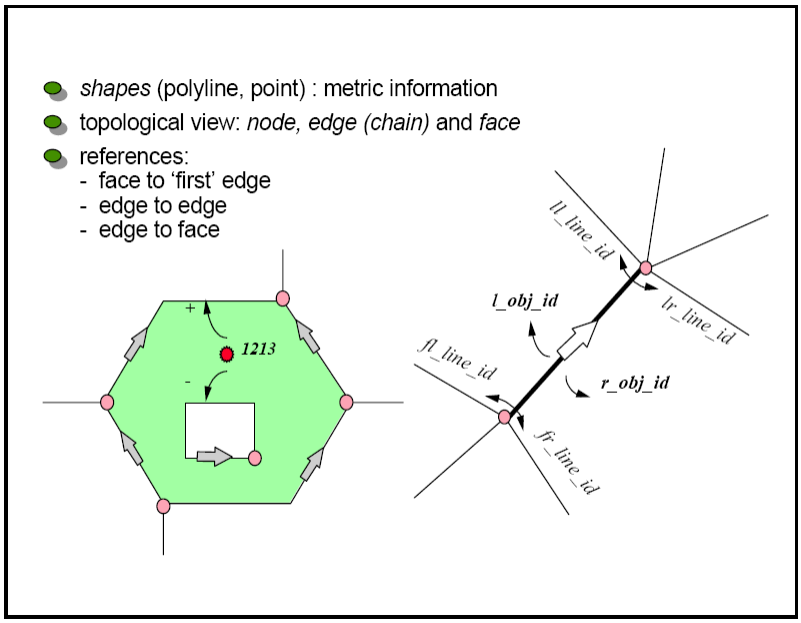


Figure 2.2 The winged edge model used by the Dutch Cadastre (taken from: Quak, Stoter & Tijssen 2003)

2.2.2. Oracle's topological structure

Oracle has developed a 2D topological structure (SDO_TOPO_GEOMETRY), which is also based on the winged edge structure. It uses three primitives: nodes, edges and faces (Kothuri, Godfrind & Beinat 2007, appendix C). Each primitive has its own table, whereas the edge is the crucial table within this structure. A node is described by its point geometry (SDO_GEOMETRY), an edge is described by its start- and endnode and by its linestring geometry (SDO_GEOMETRY)[7]. A face can be derived from the edges (a face has a reference to one edge per ring on its boundary), because every edge has a reference to its next and previous left and right edges and its left and right faces. Furthermore the MBR (minimum bounding rectangle) of each face is stored explicitly (SDO_GEOMETRY).

[7] The geometry of an edge (a line) could consist of more points than the start- and endnode (vertices are included in the geometry).

The topological tables and columns, including (data types) and primary keys of Oracle's 2D topology (sdo_topo_geometry)

NODE: node_id (number); edge_id (number); face_id (number); geometry (sdo_geometry)
EDGE: edge_id (number); start_node (number); end_node (number); next_left_edge (number); prev_left_edge (number); next_right_edge (number); prev_right_edge (number); left_face (number); right_face (number); geometry (sdo_geometry)
FACE: face_id (number); boundary_edge (number); island_edge (list of numbers); island_node (list of numbers); MBR_geometry (sdo_geometry)

The orientation of the edges is explicitly stored by its start- and endnode. The orientation of the faces is stored at edge level by the explicitly stored 'next-left/right-edge' and 'previous-left/right-edge', complemented by the storage of the left and right face of each edge. The edges are linked to a face by one (random) edge on the boundary (the explicitly stored 'boundary_edge'). Furthermore, for each primitive geometry is explicitly stored (for the face the minimum bounding rectangle is used) and isolated nodes and edges in faces, are explicitly stored in both the node and face table (Murray et.al. 2008b).
There is quite some redundant storage within this structure. The most obvious redundancy is at the geometry and at the references. First, both previous and next edges are stored. Second, island nodes are stored in both the node and face table. Island edges could easily be derived from the edge characteristics (when the left and right face are the same) but are also stored at face level. Third, the node-edge relationship is stored at both node (only one reference to a random edge it is bounding) and edge level and the edge-face relationship is stored at both edge and face level (only one reference to a random edge which the face is bounded by). These redundancies are introduced for reasons of performance, which will be better. A disadvantage of storing redundancies is more storage and a greater chance for contradictions, which need to be tested in a validation test. Oracle has implemented a validation function for this topological structure, which will be described in chapter 4.[8]

Because Oracle is used for implementation in this research, a few details of the technical model will follow. Every table has a primary key on the primitive's id. Every other field (except for the primary keys) is allowed to have null values. The topology tables are linked to feature tables via the relation table by relating the primitive types (node-point, egde-line, face-polygon) and their id's (but this is out of scope of this research).

PostGIS is the spatial extension of the open source PostgreSQL DBMS. Its topology support is in a pre-alpha stage and is very much the same as Oracle's topology. Both structures have three primitives and three corresponding tables, are based on the winged edge structure and store geometries at node and edge level and the MBR for the faces. There are three main differences. First, PostGIS has less redundancy in its references: in the face table there is no reference to a random edge (on its boundary) and in the node table there is no reference to a random edge (as start- or endnode). Second, PostGIS has only one way references for the winged edges stored, the next left/right edges are stored explicitly, the previous edges are not stored. Finally PostGIS has less singularities explicitly stored, only node-in-face (only within the node table) and not edge-in-face (http://trac.osgeo.org/postgis/: consulted May 2010).

2.2.3. Radius Topology (1Spatial)

Radius Topology is an extension for Oracle spatial database. The approach is different from Oracle's topology model. Radius Topology stores all topological references explicitly, in separate tables, which leads to many more tables. The primitives node, edge and face are stored in the NODE, EDGE and FACE tables and all topological references are stored separate in linking tables, including the winged edges in the EDGE_TO_EDGE table. A

[8] SDO_TOPO_MAP.VALIDATE_TOPO_MAP and SDO_TOPO_MAP.VALIDATE_TOPOLOGY.

TOPO table links the features and the topology, similar to Oracle (Quak, Stoter & Tijssen 2003).

The topological tables and columns, including (data types) of 1Spatial's 2D topology (Radius Topology)

NODE: node_id (number); face_id (number); x (number); y (number);
EDGE: edge_id (number); geometry (geom);
FACE: face_id (number);
EDGE_TO_NODE: edge_id (number); flag (Boolean); node_id (number);
EDGE_TO_EDGE: edge1_id (number); flag (boolean); edge2_id (number);
LINE_TO_EDGE: line_id (number); position; edge_id (number); flag (Boolean);
FACE_TO_EDGE: face_id (number); edge_id (number); flag (Boolean);
AREA_TO_FACE: area_id (number); face_id (number);

Recently 1Spatial has developed an ISO(19107) compliant system and method for 3D topology (Watson, Martin & Bevan 2008). Each primitive (node, edge, face and volume) has its own table and all nodes, edges and faces have their geometry explicitly stored. The singularities node-in-face, node-in-solid and edge-in-solid are also explicitly stored. The relationships between the primitives are stored in linking tables (like in the 2D version) including an orientation: NODE2EDGE, EDGE2FACE and FACE2SOLID. The topology is linked to features by four tables (one for each primitive).

The topological tables and columns, including (data types) and primary keys of 1Spatial's 3D topology (Radius Topology)

NODE: id (number); realization (geometry); isolatedinface; isolatedinsolid;
EDGE: id (number); realization (geometry); isolatedinsolid;
FACE: id (number); realization (geometry);
SOLID: id (number);
FACE2SOLID: face_id (number); solid_id (number); orientation (Boolean);
EDGE2FACE: edge_id (number); face_id (number); orientation (Boolean);
NODE2EDGE: node_id (number); edge_id (number); orientation (Boolean);

2.2.4. 3D FDS

3D FDS (Formal Vector Data Structure) of Molenaar is defined by a conceptual model and 12 conventions. The conceptual model consists of three levels: features (related to a thematic class), geometrical primitives (point, line, body and surface) and topological primitives: node, arc, face and edge. The edge primitive is constructed by a number of arcs (which are defined by a start- and endnode). The sequence of arcs (plus its direction) determines the orientation of a face. No 3D topological primitive is used in this structure. The geometrical solid can be composed from the topological faces (the relationship solid-face is explicitly stored in the FACE table, including the orientation). Explicitly defined singularities are the topological relationships node-in-face and arc-in-face, complemented with the relationships between topological and geometrical primitives node-in-solid and arc-in-solid (Zlatanova 2000). Rikkers et al. (1993) proposed mapping the model into a relational database, see the following frame for details.

The topological tables and columns, including (data types) of Molenaar's 3D FDS by Rikkers et al.

NODE: node_id (number); geometry (geom);
ARC: arc_id (number); begin_node (number); end_node (number);
EDGE: face_id (number); sequence_arcs (list of numbers); arc_id (list of numbers); direction (boolean)
FACE: face_id (number); face_part_of_surface (number); body_left (number); body_right (number)

2.2.5. SSS

The Simplified Spatial Schema (SSS) of Zlatanova (2000) was designed for visualization purposes and uses only two primitives (node and face). No 3D primitive is maintained, but defined by faces. The geometry is stored at node level, edges and faces are described by the order of nodes, and surfaces and volumes are described by the order of faces. The orientation of the primitives is stored by the sequence of nodes or faces.

The topological tables and columns, including (data types) of the Simplified Spatial Schema (Zlatanova)

NODE: node_id (number), geometry (geom);
EDGE: edge_id (number); sequence_nodes (list of numbers); node_id (list of numbers);
FACE: face_id (number), sequence_nodes (list of numbers); node_id (list of numbers);
SURFACE: surface_id (number); sequence_faces (list of numbers); face_id (list of numbers);
VOLUME: volume_id (number); sequence_faces (list of numbers); face_id (list of numbers);

2.2.6. TEN

The TEN (tetrahedral irregular network) was introduced by Pilouk (1996). The TEN has a real 3D primitive (tetrahedron) involved. In the TEN structure each volume is a tetrahedron (existing of four faces), each face is a triangle (existing of 3 edges) and each edge is an arc (existing of 2 nodes). Features exist of a number of tetrahedra. The structure covers a full 3D space partition and no singularities are permitted; no dangling or isolated primitives are allowed. In the relational implementation, three topological tables are defined: TRIANGLE, ARC and NODE. The geometry is stored at node level, the arc-node relationship is stored in the ARC table and the tetrahedron-triangle-arc relationship in the TRIANGLE table. The orientation of the edges is implied by their start- and endnode, the orientation of the faces (triangles) is stored by defining three ordered edges. Each face has a tetrahedron on each side.

The topological tables and columns, including (data types) of the TEN-structure (relational implementation)

NODE: node_id (number); geom (geometry);
ARC: arc_id (number); begin_node (number); end_node (number);
TRIANGLE: triangle_id (number); tetra1 (number); tetra2 (number); edge1 (number); edge2 (number); edge3 (number);

Penninga (2008) used the TEN structure based on the simplicial homology. By storing a geometry-code for each tetrahedron, (constrained) triangles, (constrained) edges and nodes could be derived (in views).[9]

In the next section a conclusion will be drawn on the discussed topological structures. And the characteristics for a topological structure will be defined. In the next chapter (chapter 3) those characteristics will be filled in and a conceptual model will be designed

2.3. Conclusion

Many 2D topological structures are based on the winged edge structure, like the discussed examples. When jumping to the third dimension some difficulties arise. First, there is the amount of primitives involved and relationships between the primitives which is higher in the third dimension, but not only more relationships, also more complex relationships exist in 3D. Furthermore the orientation is far more difficult, e.g. the orientation of an outer ring of a face cannot simply be defined as counter clockwise anymore, but the orientation is now related to the volume the face is bounding. Each face has two orientations (negative and positive) and an edge could be on the boundary of more than 2 faces, which is not possible in 2D. This leads to a complex winged edge

[9] This structure is not elaborated any further.

structure in 3D. In 2D an edge has 4 'wings' (one previous left, one previous right, one next left and one next right) and 2 faces (one left face and one right face). In 3D an edge could have more than 4 'wings' and also more than 2 faces. Even left and right are not unambiguous anymore in 3D.
Several 3D topological structures have been discussed, some main differences between those structures will be summarized. There is a difference in maintaining primitives, e.g. the TEN maintains a 3D primitive, while 3D FDS does not. There is also a difference in the number of used primitives, SSS only maintains two primitives (nodes and faces), while Radius Topology maintains four primitives (node, edge, face and volume). Another difference is concerning the space partition, SSS does not require a full space partition while a TEN does assume a full space partition. Furthermore the explicit stored singularities differ between the structures, while a TEN does not permit any singularities, 3D FDS stores the singularities node-in-face, node-in-solid, arc-in-face and arc-in-solid explicit. Orientation is stored in different ways as well, sometimes a sequence is used, for example in SSS the sequence of the nodes in a face determine the orientation of the face, while in Radius Topology the orientation is stored explicitly by a sign (positive or negative).
There also some differences at implementation level, within the number of tables and columns, the used data types, but also the redundancy. For example, the TEN structure has only three tables, while Radius Topology has seven tables.

As said at the beginning of the previous section, topological structures are mainly determined by a only few main characteristics: space/plane partitioning, used primitives, constructive rules, orientation and explicit/implicit relationships. From the presented overview of the topological structures it can be concluded that there are many possibilities in designing a topological structure. Since there is no single 3D topological structure best suitable for all types of applications, it is very important to define the context and requirements of the topological structure first. The context of this research is a topological structure within a DBMS in the third dimension. Six aspects which has to be taken into account when designing a topological structure have been distinguished:

- *Dimension*

Which dimension will the structure maintain

- *Partition (including the universe)*

The structure will either present a partition or a network. When presenting a full space or volume partition, 'everything' is modelled (including air). Which means no gaps and overlaps are allowed. A choice has to be made on how to model the universe of the structure.

- *Primitives (including the (co)boundary relationships)*

A choice has to be made on which topological primitives to use (node, edge, face and/or volume) and how are they related to each other?

- *Orientation*

A special characteristic of a topological primitive is its orientation. How and where to store the orientation or maybe a structure can do without? Different possibilities exist, as shown in the previously discussed topological structures.

- *Singularities*

Which singularities are permitted within a structure?

- *Geometrical realization*

The geometry has to be stored at one place at least. A choice has to be made on where and how to store the geometry, in order to maintain the relationship between single geometries and the topological structure. Furthermore a choice has to be made on the kind of geometry, will the structure be linear or are curves allowed as well?

The requirements for the topological structure will be set in the next chapter (chapter 3), based on the above aspects. When the requirements are set, validation rules are defined in chapter 4, in order to be able to maintain the structure. With the implementation (chapter 5) a choice is made on which relationships, which are defined in the requirements, are stored explicitly within the structure. Storing all topological

relationships requires a high amount of storage space and leads to a lot redundancy. On the other hand storing few relationships will lead towards a lot of computing.

3. Design of a 3D topological structure

In the last section (3.2) of this chapter the conceptual model of the topological structure will be presented, but first the requirements for the topological structure will be appointed in section 3.1. In the same section (3.1) the standard ISO 19107 will be discussed. The conceptual model will be implemented in Oracle in chapter 5. In between, in chapter 4, validation rules for the structure will be set.

3.1. The requirements for a 3D topological structure

As stated in the scope of this project, a 3D topological structure within a DBMS will be created. The context of the structure in three keywords is: 3D, topological structure and DBMS. As mentioned in section 2.2, there is no single topological structure best suitable for all types of applications. Therefore some additional conditions will be added to the context.
The structure which will be designed will represent a full partition of space. This means, in geometrical terms, 'no gaps and overlaps allowed'. Since the structure is 3 dimensional it will consist of volumes. The volumes are represented by their boundaries and consist of faces, which consist of edges, which consist of nodes. Furthermore the structure will be linear[10] and the extent (universe) of the structure will be determined by the applicable coordinate system. In the Netherlands this will be the Dutch 'Rijksdriehoeksmeting' for the horizontal (xy) system and 'NAP' for the vertical (z) system. Finally the structure will be ISO 19107 compliant.

Context	
• Data definition • Storage • Dimension	• Topological structure • DBMS • 3D
Conditions	
• Partition • Representation • Primitives • Extent • Linearity • Standard	• Full space partition • Boundary representation • Volumes, faces, edges and nodes • Applicable coordinate system • Linear • ISO 19107 compliant

Table 3.1 The context and additional conditions for the topological structure

Next the requirements for the topological structure will be defined, based on the six aspects listed in section 2.3: dimension, partition (including the universe), primitives (including the (co)boundary relationships), orientation, singularities and geometrical realization. The standard ISO 19107 (ISO/TC211 2003) provides a conceptual spatial schema that deals with spatial characteristics and operators. Based on this spatial schema the characteristics of a topological structure will be discussed and the requirements for the topological structure will be set indicating what is valid and what is not valid. The dimension is 3D, the other five characteristics will be discussed in the following order: orientation, boundaries (primitives), singularities, the universal volume (partition) and geometrical realization.

3.1.1. Orientation

According to ISO 19107, a topological primitive is a single, non-decomposable element with spatial characteristics that are invariant under continuous transformations. Four topological primitives are distinguished: the node (TP_Node), edge (TP_Edge), face (TP_Face) and volume (TP_Volume[11]). Each primitive is associated with two directed

[10] Planarity / Linearity: Planarity is used for 'lying in one plane', this can only apply to faces (volumes never lie in one plane and straight edges and nodes always lie in one plane). When a face is planar it is meant to be flat (within a certain tolerance). When a volume consists of only planar faces it is called linear.
[11] A topological solid and TP_Solid by ISO, but the terms 'volume' and TP_Volume will be used in this research.

primitives. A directed topological primitive (TP_DirectedTopo) represents the logical association between a topological primitive and one of its two orientations.

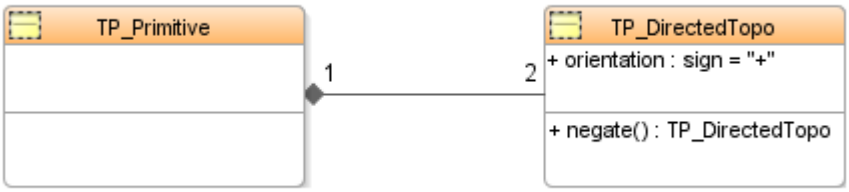


Figure 3.1 The association between topological primitives and their directed primitives, according to ISO 19107

By convention, a topological primitive is equivalent to its positive directed topological primitive (only positive oriented directed primitives are stored). Orientation plays an important role within a topological structure. In the third dimension, orientation is more complex than in the second dimension.
The orientation of a 3D face can not be specified as clockwise or counter-clockwise like in 2D. In the third dimension a face has a positive and a negative side. Which is used to distinguish the interior (negative) from the exterior (positive) of a volume. The orientation is specified by the normal of a face, which can point outward (positive) or inward (negative) from the volume it bounds. The rule is that a normal should always point outward the volume.
The orientation of an edge in 3D can be defined as positive or negative, like in 2D. The orientation represents the positive and negative direction along an edge, which is determined by its start- and endnode. However, the difference between 2D and 3D is the number of faces an edge can bound. In 2D an edge has always one left and one right face. In 3D, there is no left and right anymore and an edge can be part of more than two faces. Edges of a face are ordered within rings, forming a cycle. There is no rule for the orientation of rings, except that inner rings should be directed in the opposite direction of the outer rings.
A directed node represents the orientation of a node. The direction is, with respect to an edge, positive for an endnode and negative for a startnode.

3.1.2. Boundaries

Within the boundary representation, the boundary of a volume is defined by a collection of faces forming one or more shells. This means that the interior and exterior of the volume are automatically defined (explicitly or implicitly).
A boundary is a set (an unordered collection with no repetition) of directed primitives (in a cycle), which represents the boundary of a primitive. The dimension of a boundary is always one less than the dimension of the original primitive. The boundary relationship is bidirectional, the reverse relationship defines the coboundary. The coboundary is also an (unordered) set of directed primitives, which represents all the primitives that have this particular primitive on their boundary. The dimension of a coboundary is always one higher than the dimension of the original primitive.
According to ISO the boundary is structured as TP_PrimitiveBoundary, with the attributes exterior and interior or start- and endnode.[12] The boundary of an edge is defined by two directed nodes, the start- and endnode, but for the boundaries of volumes and faces special data types are defined (TP_Ring and TP_Shell). A ring (TP_Ring) consists of a sequence of directed edges connected in a cycle and is used to represent a single boundary of a face. The endnode of each directed edge in the sequence is the startnode of the next directed edge in the sequence. A shell (TP_Shell) consists of an (unordered[13])

[12] TP_PrimitiveBoundary is a dataType, used to structure the boundaries in a convenient manner.
[13] Unlike rings which exist of an ordered sequence of edges.

set of directed faces connected in a topological cycle and is used to represent a single boundary of a volume.

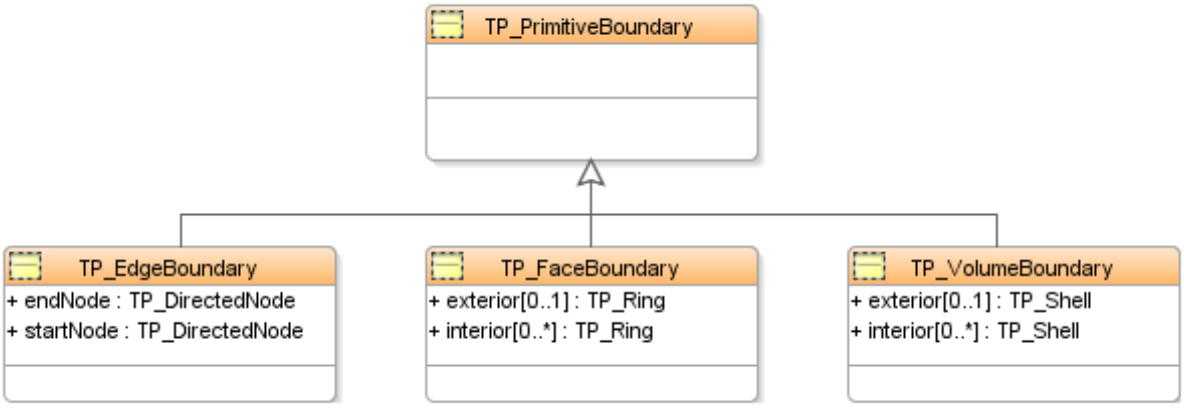


Figure 3.2 The boundaries of the topological primitives according to ISO 19107

Inner rings create holes and inner shells create voids. Three kind of voids could be distinguished; a void, a tunnel and an atrium. A void is completely inside the volume (figure 3.3), a tunnel goes 'through' the volume (figure 3.3) and an atrium is a kind of a pit in a volume (figure 3.4). All three kinds of voids are allowed. The 'pure' void is modelled with an outer shell and an inner shell, the tunnel and atrium are modelled with only an outer shell.

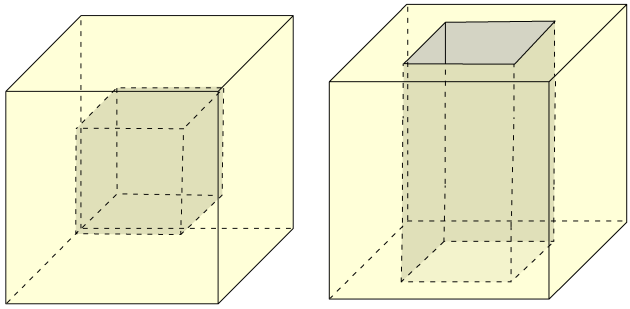


Figure 3.3 A void and a tunnel

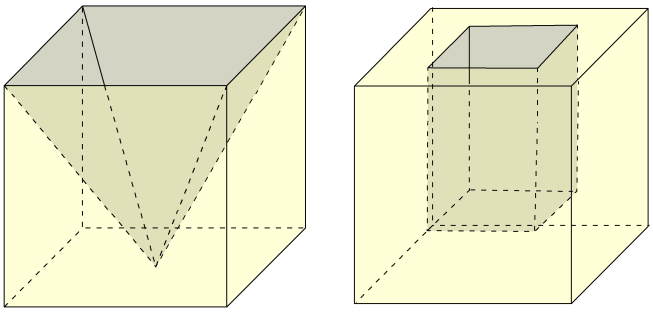


Figure 3.4 Two atriums

Inner shells are allowed to touch their outer shell by node or edge (not by face) and inner rings are allowed to touch their outer ring by node (not by edge). Shells are allowed to touch themselves, while rings are not allowed to touch themselves according to the Simple Feature Specification (Ryden 2005). Though it would actually be a valid situation when a ring touches itself in a node (if the area remains contiguous), this research will stick to the SFS rule 'a ring is not allowed to touch itself'. This is illustrated in figure 3.5. A tunnel is modelled by a complex outer shell, which touches itself (one

edge is used four times).[14] This is a valid situation as long as the top- and bottom faces are modelled as one outer ring with one inner ring, touching each other. For faces it is not allowed to touch themselves and 'faking' a hole.

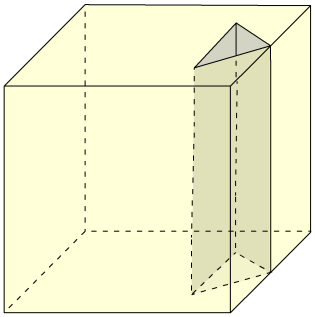


Figure 3.5 A complex outer shell

The boundary relationships between the four primitives can be defined in more detail, by adding their cardinality. Each volume consists of one outer shell and zero or more inner shells. Each shell of a volume consists of four or more faces. Each face consists of one outer ring and zero or more inner rings. Each ring of a face consists of three or more edges. Each edge consists of one startnode and one endnode.

There is not only a boundary relationship (is bounded by), but this association is bidirectional, which means the coboundary (is the boundary of) relationship is also present.
Because a full space partition is present, the structure only allows volumes. Faces, edges and nodes are only present as boundary primitives of the volumes. This means no dangling and isolated primitives are allowed. Isolated primitives are primitives not connected to a primitive one dimension higher, like nodes which are not connected to an edge, edges which are not connected to a face and faces which are not connected to a volume. Dangling primitives are primitives which are connected to one or more primitives one dimension higher, but are not part of its boundary. Like edges which are connected to a face but are not within the ring that bounds the face or faces which are connected to a volume, but which are not part of the volume's shell.[15] In figure 3.6 an example of an isolated and dangling edge is displayed.

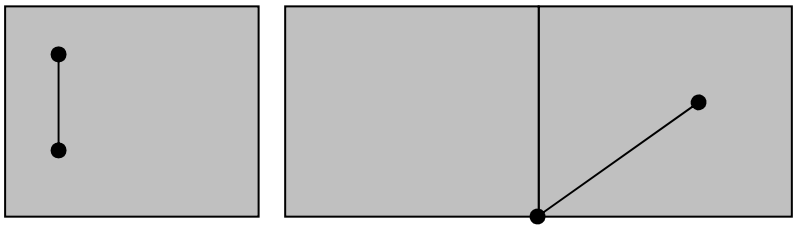


Figure 3.6 An isolated edge (left) and a dangling edge (right)

Because isolated and dangling primitives are not allowed in the structure and the full partition of space, every volume has a neighbour on each side. This means every face is on the boundary of exactly two volumes. For edges the situation is slightly different, because an edge is on the boundary of at least two faces, but possibly on the boundary of more than two faces. The same applies for nodes, which can be on the boundary of three or more edges. For the primitives on the boundary of the structure, other rules apply, this will be dealt with in section 3.1.4 (universal volume). The (co)boundary relationships between the primitives are summarized in table 3.2.

[14] A tunnel (or atrium) could never be modelled by an inner shell.
[15] Nodes and volumes can not be dangling.

	Boundary relationship	Coboundary relationship
Node-edge	one edge is bounded by two nodes	One node is on the boundary of three or more edges
Edge-face	one face is bounded by three or more edges	One edge is on the boundary of two or more faces
Face-volume	one volume is bounded by four or more faces	One face is on the boundary of two volumes

Table 3.2 The (co)boundary relationships between the topological primitives

From these relationships other relationships between primitives which differ more than 1 dimension can be derived. For example the node-face relationship. Each face consists of three or more nodes and a node is part of three or more faces. In the same way the relationships edge-volume and node-volume can be derived. These 'part of' relationships will not be modelled because they can easily be derived from the (co)boundary relationships. Special situations within this these relationships will be modelled as singularities, which will be discussed in the next section.

3.1.3. Singularities

Singularities refer to primitives which are related to each other without an intermediate primitive. In a structure with four primitives 6 singularities can be distinguished:

- node-in-edge
- node-in-face
- node-in-volume
- edge-in-face
- edge-in-volume
- face-in-volume

The singularities, node-in-volume, edge-in-volume and face-in-volume, represent isolated or dangling primitives and are not allowed in this structure. The singularity, node-in-edge, represents a 'vertex' on an edge, which is not allowed in this structure. In case of a node-in-edge singularity the edge should be split in two separate edges, see figure 3.7.

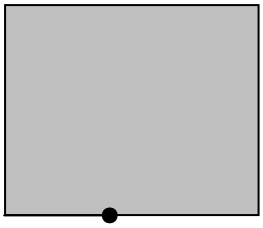


Figure 3.7 A node-in-edge singularity (invalid),
the bottom edge should be modelled as two separate edges

The singularity node-in-face represents nodes, which are related to a face, but are not attached to an intermediate edge on the boundary of that face. This singularity is valid as long as it satisfies the (co)boundary relationships. When it does not satisfy the (co)boundary relationships, the node is isolated and not valid. The same applies for the singularity edge-in-face. The singularity edge-in-face represents edges, which are related to a face, but which are not part of the boundary of that face. The edge-in-face singularity is valid as long as it satisfies the (co)boundary relationships. Note the difference between the two singularities and isolated primitives in figure 3.8 and figure 3.9. In both situations there is a node-face and an edge-face relationship without an intermediate primitive. The difference however is that in figure 3.8 the highlighted nodes and edge satisfy the (co)boundary relationships, they are part of another face, and are valid as singularities node-in-face and edge-in-face. While in figure 3.9 the highlighted nodes and edge are not part of any face and therefore do not meet the (co)boundary criteria, which is invalid.

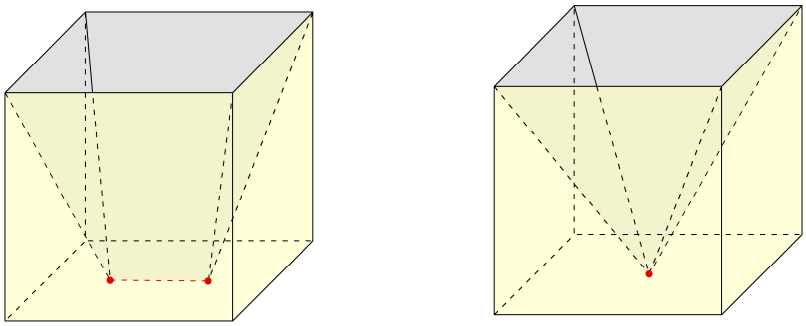


Figure 3.8 Singularities edge-in-face and node-in-face (allowed)

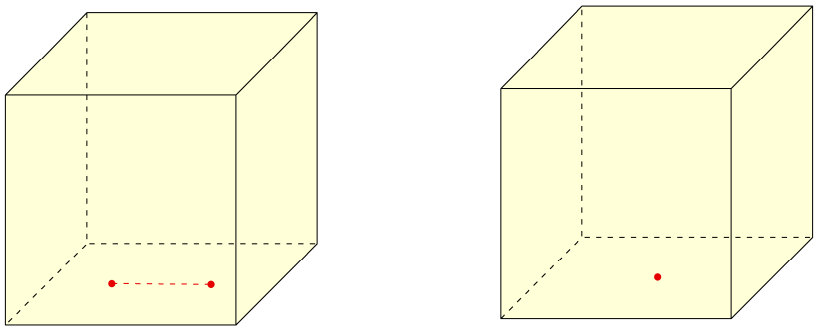


Figure 3.9 Isolated edges and nodes (not allowed)

The singularity edge-in-face could be anywhere in the face, even on the border of the face, as long as the particular edge does not split the face in two parts. This means that not both start- and endnode can be on the boundary of the face. This is illustrated in figure 3.10. It shows that the left and middle figures are valid edge-in-face singularities, assumed that the edges are connected to other (perpendicular) faces (otherwise they are isolated or dangling). While the right figure is not a singularity anymore, it is a 'normal' edge which is on the boundary of (at least) two different faces.

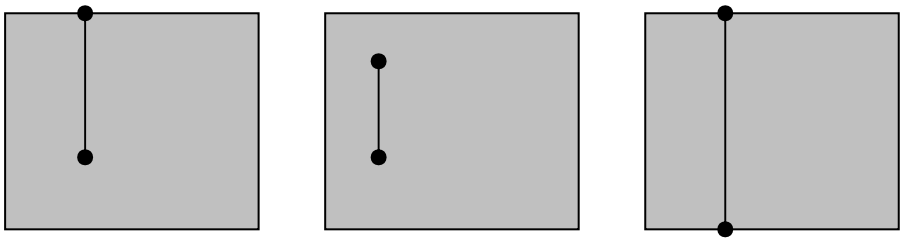


Figure 3.10 Edge-in-face singularities (the right figure is a 'normal' edge)

3.1.4. Universal volume

The universal volume is an unbounded topological volume in a 3D complex. The universal volume is normally not part of any feature and is used to represent the unbounded portion of the data set by its inner shells. The universal volume could have more than 1 inner shell, so more 'universes' are possible. An universe can not have voids. The solution will be modelling the void as another volume.

3.1.5. Geometrical realization

The relationship between the topological primitive (TP_Primitive) and geometrical primitive (GM_Primitive) is defined by the association 'realization' in ISO. The geometrical primitive is the geometrical realization of the topological primitive (and vice versa).

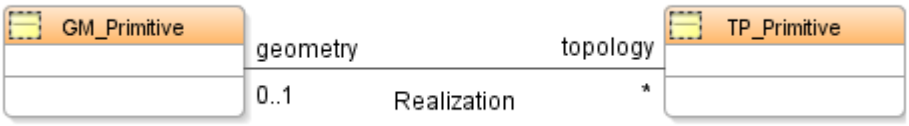


Figure 3.11 The association between geometrical and topological primitives according to ISO 19107

A geometrical primitive can be realized by zero or more topological primitives, while a topological primitive can be realized by zero or one geometrical primitives. This difference is caused by complexes. ISO distinguishes, next to the primitives, a topological complex (TP_Complex) and a geometrical complex (GM_Complex[16]). A complex is an organized structure of primitives (of different dimensions up to the dimension of the complex). The primitives and complexes are linked by the association 'complex'. Every topological primitive must be in at least one topological complex. However, a geometrical primitive does not have to be part of a geometrical complex, it can stand alone.

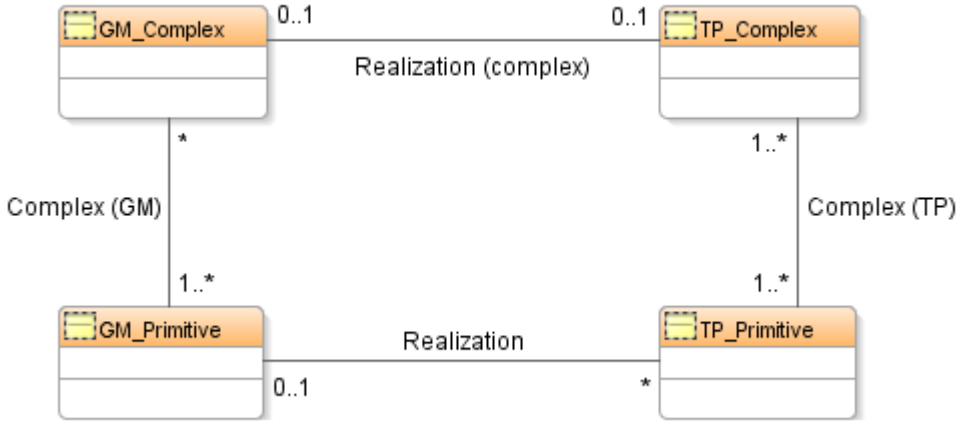


Figure 3.12 The association between geometrical and topological primitives and complexes, according to ISO 19107

The association 'realization' links the TP_Primitive to the GM_Primitive. When a geometric complex is the realization of a topological complex, then the primitives in each shall be in a dimension-preserving 1-to-1 correspondence. Since a topological primitive is in any (realized) topological complex, it will be associated to exactly one geometrical primitive. Except when a topological primitive is used to describe a logical topological structure that is not realized by a geometrical complex, then this relationship will be empty for all topological primitives contained in this topological complex. A single geometrical primitive may be involved in many independent geometrical complexes, each of which may be the realization of different topological complexes. Thus, a geometrical primitive may be the realization of many different topological primitives. It is possible that a particular instance of a topological primitive is realized by a geometrical complex of the same dimension. In that case the realization is empty.
Within this structure the geometrical realizations are defined by separate operations for each topological primitive. The geometry will be stored at node level, to avoid redundancy.

The requirements, as listed in section 2.3, have been set. They are summarized in table 3.3. In the next section a conceptual model will be designed based on these requirements.

[16] GM_Complex is not explicitly implemented by the International Standard 19107.

Requirement	
Dimension	3D
Partition (the universe)	full space partition. The universe is modelled by one or more inner shells of the universal volume.
Primitives (boundary relationships)	The structure exists of: volumes, faces, edges and nodes. Boundary: The boundary of each volume consists of one outer shell and zero or more inner shells. A shell is a set of 4 or more faces. The boundary of each face consists of one outer ring and zero or more inner rings. A ring is a set of 3 or more edges. Each edge is bounded by exactly two nodes. Coboundary: one node is on the boundary of three or more edges, one edge is on the boundary of two or more faces and one face is on the boundary of two volumes. No isolated and dangling primitives allowed.
Orientation	Each primitive is associated with two directed primitives
Singularities	two singularities are allowed: node-in-face and edge-in-face
Geometrical realization	The structure is linear with planar faces and straight edges. Each topological primitive is linked to a geometrical primitive by the association 'realization'. Geometry itself is stored at node level.

Table 3.3 Summary of the requirements for the topological structure

3.2. The conceptual model

In order to design a topological structure, the main characteristics of the structure need to be translated into a model. The conceptual model consists of classes with relationships, attributes and operations. The applicability of the structure depends also on the availability of spatial operations (Verbree & Zlatanova 2004). For this research only operations for geometrical realizations will be developed.

First a very thin model will be presented, where only the core aspects are included. Then the model will be extended with the other characteristics.
The core aspects are the four primitives with their boundary relationships and the relationship with the geometries. The structure consists of only volumes, which are bounded by faces, which are bounded by edges, which are bounded by nodes. For each primitive a geometry operation is added and the geometry itself is stored at node level (figure 3.13).

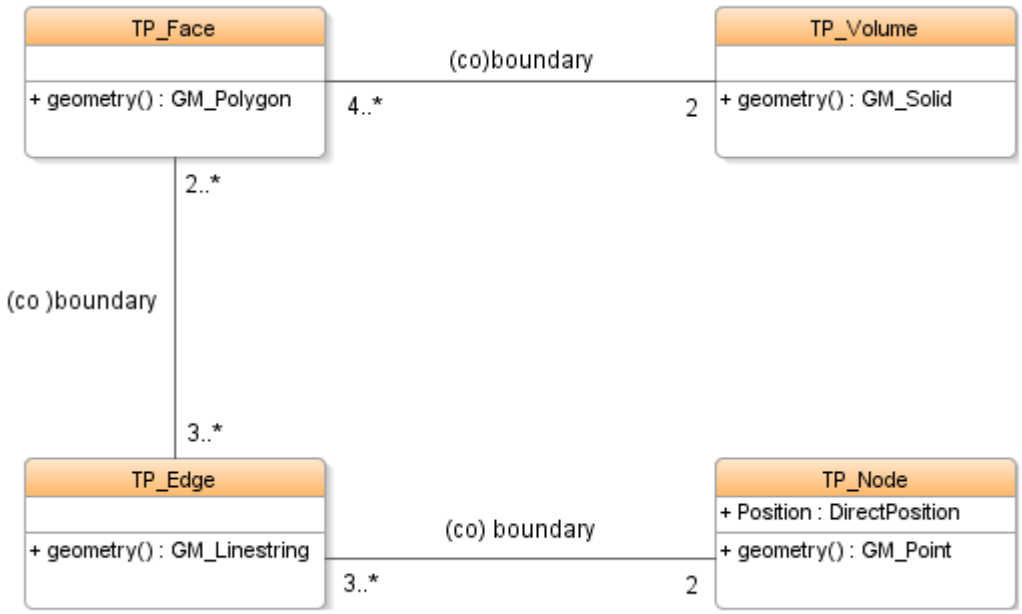


Figure 3.13 Conceptual model, the thin version

This thin version of the conceptual model is very simple and robust. Although the core aspects are modelled, other aspects, like orientation, which are very crucial within a topological structure, are not within the model. Therefore the model is extended. The main characteristics of the topological structure, discussed in section 3.1, are admitted into the conceptual model.

Partition (the universal volume): The cardinalities specified in the (co)boundary relationships imply a full space partition. Within the conceptual model the universal volume is not specifically modelled, although the specified cardinalities of the boundary relationships require a universal volume.
Primitives (boundary relationships): Each of the four primitives is modelled as a separate class, related to each other by the (co)boundary relationships. Which are modelled as bidirectional associations between the primitive classes. The cardinality of these relationships imply that no isolated and dangling primitives are allowed. The face and volume boundaries, rings and shells, are added to the boundary associations as association classes.
Orientation: Three association classes are admitted on the boundary associations, to model orientation. These three classes represent the orientation of nodes in edges, edges in faces and faces in volumes.
Singularities: Both node-in-face and edge-in-face are modelled as two separate (bidirectional) associations.
Geometrical realization: For each topological primitive an operation 'geometry' is added to the class. Geometry itself is stored at node level to avoid redundancy.

The requirements, as set in section 3.1, are incorporated within a transparent and widely applicable model. The conceptual model is based on the primitives and their relationships, extended with the main characteristics. Within the extended conceptual model four classes, five association classes, five (bidirectional) relationships, one attribute and four operators are defined. The main difference between the thin model and the extended model is the modelling of the orientation, the shells and rings and the singularities. On the next page the extended version of the conceptual model is presented (figure 3.14).
In chapter 5 the conceptual model will be translated into a technical model, but first validation will be discussed in the next chapter.

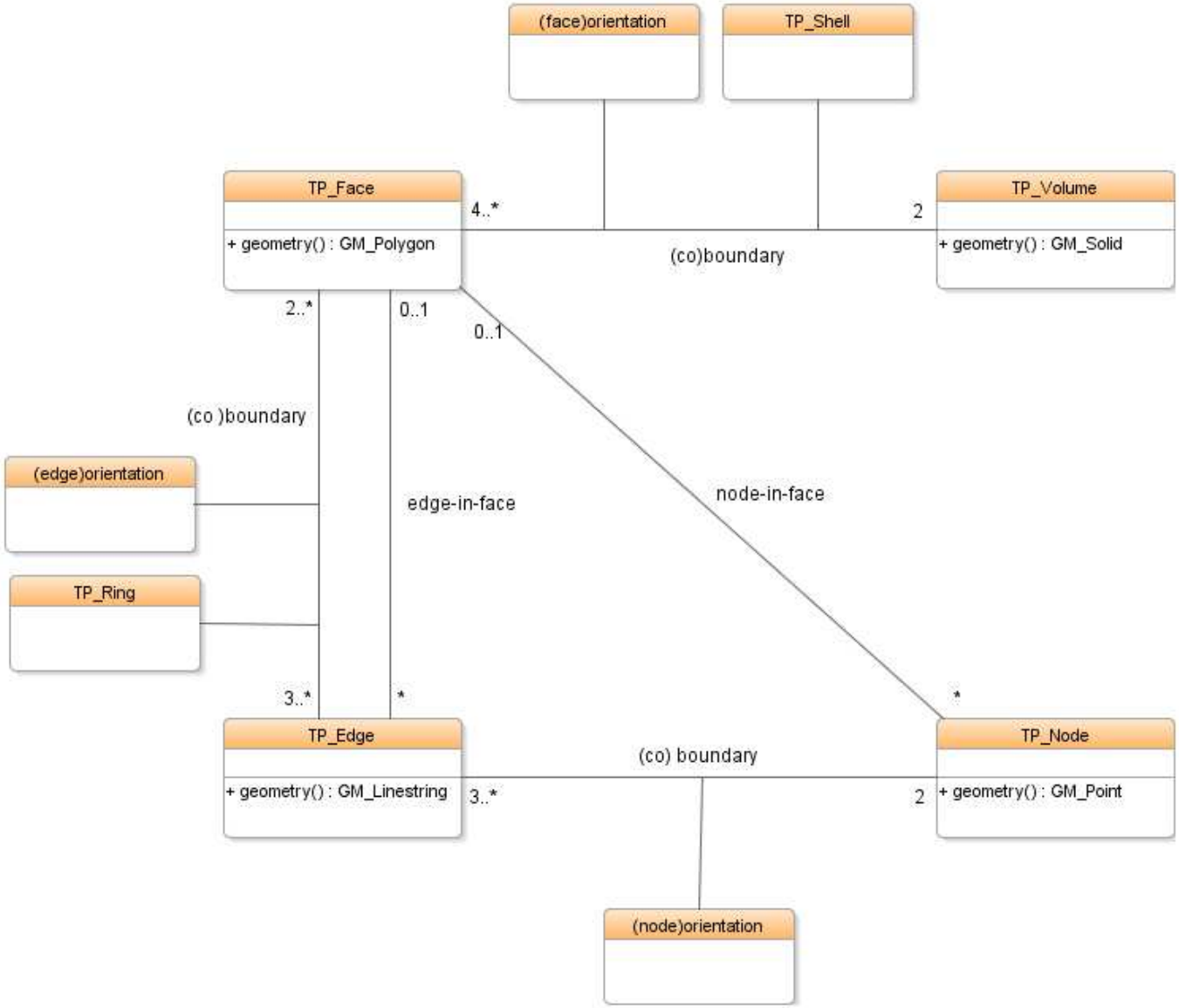


Figure 3.14 Conceptual model, the extended version

4. Validation of the 3D topological structure

In this research a validation function for the 3D topological structure, will be designed and implemented (on the designed topological structure). In this chapter, validation rules for the topological structure as defined in the conceptual model (chapter 3) will be set. First, background on validation (section 4.1) and examples of existing validation functions (section 4.2) will be discussed. Followed by the definition of the validation rules in section 4.3. In chapter 5 the validation rules will be translated into tests and implemented on the topological structure.

4.1. Introduction

Within a topological structure an important aspect, before performing any spatial analysis, is to check if the structure is valid. When a suitable valid topologically defined structure is at hand, one can start performing spatial analysis with the object(s).

Many commercial geo-DBMSs and GISs support validation of 2D single geometries. Only a few examples of validating 3D single geometries and validating topological structures are at hand. In section 4.2 some examples of existing validation functions will be presented. Due to the lack of existing validation functions of topological structures both validation functions based on single geometries (3D) and validation functions based on topological structures (2D and 3D) will be discussed.
Validation functions are characterized by the moment of validation, the user influence and the validation rules. A validation can take place when running a function, on insert (by setting constraints) or by a combination of both. A validation function can be (partly) user-defined. ArcGIS, only validating 2D geometries, is a good example of an user-defined validation function. The user can choose when to validate and what to validate (choosing from a list of predefined rules) and is able to set a tolerance value (ESRI 2009). The validation rules are the most important aspect of validation and are based on a definition of a valid primitive/structure. In case of the 3D primitive no standards are set yet. Single (2D) geometries are standardized, including implementation specifications, in the Simple Feature Specification of the OGC and ISO (Ryden 2005; ISO/TC211 2004). The Complex Feature Specification for topological primitives is not finished yet, in terms of implementation specifications. Additionally, for 3D primitives (topological and geometrical) there is an abstract ISO specification (ISO 19107:2003 *Geographic information — Spatial schema*, which has been taken over by the CEN (European Committee for Standardization) without any modifications, in 2005), but this is not the needed implementation specification (ISO/TC 2003; CEN/TC287 2005), see the frame below for some details.

Standards
The OGC standard "*OpenGIS Implementation Specification for Geographic information - Simple feature access*" exists of two parts: *Part 1: Common architecture* (OGC 05-126) and *Part 2: SQL option* (OGC 05-134)". Part 1 (common architecture) describes the common architecture for simple feature geometry (using UML notation) and implements a profile of the spatial schema described in ISO 19107:2003. Part 2 provides a standard SQL implementation of the abstract model in Part 1. The OGC standard has been published in 2005 and a candidate has been published in 2006 (part 1: OGC 06-103r3 / part 2: OGC 06-104r3). In the candidate version still only 2D geometrical primitives are discussed, but possibly within a 3D context (x,y,z coordinates). The related ISO standard (ISO 19125:2004) has been published in 2004 and is to be revised.

But even standards are not unambiguous and complete as Van Oosterom, Quak & Tijssen pointed out for 2D polygons (Van Oosterom, Quak & Tijssen 2004). Also the interpretations of different DBMS vendors differ from each other and from the standards. For topological primitives and 3D geometrical primitives there are no implementation standards.

4.2. Existing validation functions

Since not many validation functions for 3D topological structures are at hand, the validation functions, including their validation rules, for two different 3D single geometries will be discussed (including commercial and scientific sources). Arens, Stoter & van Oosterom set a definition for a 3D polyhedron of which a prototype is implemented and Oracle has set a definition for its own 3D geometrical primitive, the 'simple solid' (Arens, Stoter & van Oosterom 2005; Kazar, Kothuri, van Oosterom & Ravada 2008). Both examples have implemented a validation function. Finally two validation functions related to a topological structure will be discussed; Oracle's 2D topology validation function and the validation function of Radius Topology (3D).

Oracle has implemented a geometrical 3D primitive the 'simple solid' since the release of Oracle Database 11*g*. The definition of Oracle's 'simple solid':
'*A simple solid is a 'single volume' bounded on the exterior by one outer bounding surface and on the interior by zero or more inner bounding surfaces. To demarcate the interior of the solid from the exterior, the 3D-polygons of the outer bounding surface are oriented such that their normal vector always point 'outward' from the solid. In addition, each 3D-polygon of the bounding surfaces has only an outer boundary but no inner boundary*' (Kazar, Kothuri, van Oosterom & Ravada 2008).

Based on this definition Oracle has implemented a validation function[17] for 3D single geometries (SDO_GEOMETRY). The user can choose when to use the function and what tolerance value to use. The rules are predefined and are all checked when running the function. The validation function performs, when concerning 3D geometries, any necessary checks on the involved 2D primitives and several 3D specific checks. The function checks for type consistency[18] and geometry consistency. For the geometry consistency of a solid, the function checks for the following (Murray et al. 2008a):

- The solid must be closed.
- The solid must be connected; each face of a solid must have a full-edge intersection with its neighbouring faces, and all faces must be reachable from any face.
- An inner bounding surface must not intersect the outer bounding surface at more than one point or one line; that is, there must be no overlapping areas with inner bounding surfaces.
- No additional bounding surfaces can be defined on the bounding surfaces that make up the solid.
- All bounding surface must have a proper orientation; the vertices of all bounding surfaces must be aligned so that the normal points away from the solid.

If the geometry is valid, the function returns TRUE. If the geometry is not valid, the function returns FALSE, an Oracle error message number and the context of the error (the coordinate, edge, or ring that causes the geometry to be invalid).

Example of an Oracle error message
(a solid was validated with one non-flat (3D) polygon)

54535 Point:0,Edge:0,Ring:0,Polygon:3,Comp-Surf:1,

54535: incorrect box surface because it is on arbitrary plane, the axis aligned box surface was not on the yz, xz, or xy plane.

Kazar, Kothuri, van Oosterom & Ravada (2008) pointed out some difficulties validating Oracle's 'simple solid' and refined the validation rules:

- The volume should be contiguous: the bounding surface has to be closed and the volume has to be connected.
- Every inner bounding surface should be inside the outer bounding surface.

[17] SDO_GEOM.VALIDATE_GEOMETRY_WITH_CONTEXT or SDO_GEOM.VALIDATE_LAYER_WITH_CONTEXT.
[18] SDO_GTYPE, SDO_ETYPE, SDO_ELEM_INFO_ARRAY and DIMINFO.

- Inner bounding surfaces may never intersect, but only touch (under the condition that the solid remains connected).
- All polygons in the outer bounding surfaces are always oriented such that the normals point outward from the solid.
- Every bounding surface has to be a valid surface.
- No inner boundaries in polygons are allowed (a specific oracle implementation issue).

In addition to the Oracle validation rules the explicit rule for contiguousness has been added. A volume could be connected and closed but still not bound a single volume. This is related to the number of times an edge is used within a volume. The authors showed that an edge in a volume could be used more than two times (as long as it is an even number) whereas the volume is still valid. When this is the case the explicit test for contiguousness is needed. Now situations can occur where an edge is used more than two times (within a solid) and the solid is valid or an edge is used more than two times and the solid is invalid. See for example figure 4.1, both solids will be tested INVALID when applying the rule 'each edge in a solid must be used 2 times' and both solids will be tested VALID when applying the rule 'each edge in a solid must be used an even number of times'. When adding the rule that each solid must be contiguous the left solid will be tested VALID and the right solid INVALID.

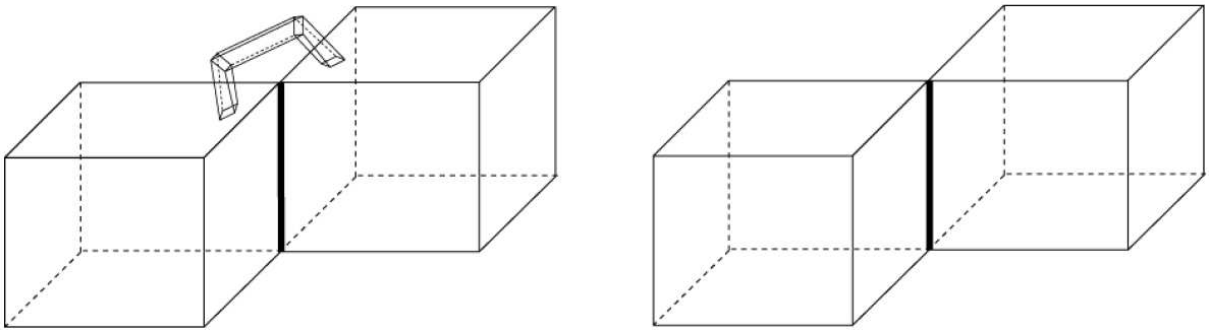


Figure 4.1 Left a valid solid; right an invalid solid
(taken from: Kazar, Kothuri, van Oosterom & Ravada 2008)

The second example of a 3D single geometry is the 'polyhedron' of Arens, Stoter & van Oosterom (2005). They set a definition for a valid (3D) geometrical solid, the 'polyhedron', and implemented a validation function (within Oracle Spatial) based on this definition.[19]

'*A polyhedron is a bounded subset of 3D coordinate space which is enclosed by a finite set of flat 3D-polygons in such a way that each edge of a 3D-polygon is shared by exactly one other 3D-polygon (Aguilera, 1998). A valid polyhedron bounds a single volume, which means that from every point (also on the bounding surface), every other point (also on the bounding surface) can be reached via the interior'* (Arens, Stoter & van Oosterom 2005).

The implemented validation function based on the above definition consists of the following tests:
- Check for consistent storage.
- Each face must be flat (within a given tolerance).
- Each polyhedron must bound a single volume in 3D space (2-manifold); vertices and edges should be 2-manifold, no intersecting faces and each polyhedron is a single object.
- Each (independent) face must be valid; they should have an area, not be self-intersecting and the inner boundaries should not intersect (touch is allowed) their corresponding outer boundaries.

[19] The authors do not agree anymore on the 2-manifold, by renewed insight (comment by P. van Oosterom).

- Each face must be oriented properly; the orientation of the vertices in the faces must be clockwise for inner shells and counter clockwise for outer shells.

These tests satisfy the used definition of a polyhedron (by Aguilera), which excludes polyhedrons like in figure 4.1 (the left solid), because of the 2-manifold rule. The authors do not agree anymore on this 2-manifold rule, because of new insights.

These were two examples of validating 3D single geometries. The goal of this research is to define a validation function for a 3D topological structure. The main difference between validating single geometries and validating topological structures is the relation between the 3D primitives. Topological primitives (volumes) are part of a whole structure, while the single geometries 'stand alone', they are validated as one object, only dealing with their internal topology and geometrical characteristics. While 3D primitives are part of a topological structure and therefore dealing with their neighbour primitives. A topological primitive must be valid on its own and valid within the structure. The explicit relationship between neighbouring primitives is one of the advantages and characteristics of a topological structure. A topological structure is not efficient when there are no (direct) neighbours around. Not many topological validation functions are at hand but Oracle has implemented a 2D topology validation function.[20] This function validates Oracle's 2D topological structure (as described in chapter 2), by checking the consistency of all pointer relationships among edges, nodes and faces and for the following conditions (Murray et al. 2008b):

- All faces are closed, and none have infinite loops.
- All previous and next edge pointers are consistent.
- All edges meet at nodes.
- Each island node is associated with a face.
- All edges on a face boundary are associated with the face.

These five conditions are all related to consistent storage and references between the primitives, but the function also checks the following conditions related to geometry:

- Each island is inside the boundary of its associated face.
- No edge intersects itself or another edge.
- Start and end coordinates of edges match coordinates of nodes.
- Node stars[21] are properly ordered geometrically.

The difference between the geometrical validation function and the topological validation function will be illustrated by the example of intersecting edges. In the topological validation function of Oracle all edges are tested for intersecting themselves or intersecting other edges (of other faces), while in the geometrical validation function polygons are tested for self intersection only. This means lines are only tested for self intersections and intersection with lines of the same polygon, but not for intersections with lines of other polygons. When checking for intersection between polygons, another function has to be used, like SDO_OVERLAPBDYINTERSECT.

Radius Topology recently implemented a 3D topological structure including a implementation method, describing how to implement a user defined geometry into the topological structure of Radius Topology (Watson, Martin & Bevan 2008).[22] Within this

[20] SDO_TOPO_MAP.VALIDATE_TOPOLOGY.

[21] The node star of a node is the edges that are connected to the node. A positive edge ID represents an edge for which the node is its start node. A negative edge ID represents an edge for which the node is its end node. If any loops are connected to the node, edges may appear in the list twice with opposite signs (Murray et al. 2008b).

[22] This is done in a few steps, the first step is 'spatial refinement', which will take place to determine general spatial relationships between the features, within a predefined tolerance range, which should be related through topology (topological primitives). In the next step a refinement will be applied on the actual interactions. This refinement will determine which topological primitives have actual spatial interaction, within the specified tolerance range, to permit them to be associated with each other. In the third step the common sections of each topological primitive that is affected by each interaction will be individually determined. Each topological primitive affected by each interaction will be checked to determine the shared geometry sections. Generally the input geometries will not marry perfectly into the created topological complex. Some snapping of the input

implementation process the topological primitives will be validated by a set of validation rules. A minimum set of rules has been pointed out:

- All solids are closed
- All faces are closed and none have infinite loops
- All previous and next edge pointers are consistent
- All edges meet at nodes
- Each isolated node is associated with a face or solid
- All faces on a solid boundary are associated with the solid
- All edges on a face boundary are associated with the face
- Each isolated edge is inside the boundary of its associated face or is isolated within a solid
- No face intersects itself or another face
- No edge intersects itself or another edge
- Start and end coordinates of edges match coordinates of nodes exactly
- Including a referential check of all pointers to exclude dangling or null pointers and other referential errors

It strikes that these rules are very similar to Oracle validation rules, which is not surprisingly since the Radius Topology is build on Oracle Spatial. All rules of the 2D topological validation function are adopted, except for the check 'node stars are properly ordered geometrically'. The Radius Topology structure does not work with previous and next edge pointers, therefore the rule 'all previous and next edge pointers are consistent' seems not really relevant. Furthermore some checks are extended to the third dimension: 'each isolated node is associated with a face or solid' (the last option is added) and 'each isolated edge is inside the boundary of its associated face or is isolated within a solid' (the last option is added). Also some checks are added for the third dimension, based on Oracle's 3D geometrical rules: 'the solid must be closed', 'all faces on a solid boundary are associated with the solid' and 'no face intersects itself or another face'.

In the next section the validation rules for a 3D topological structure will be set based on the above discussion and the conceptual model presented in chapter 3.

4.3. A valid 3D topological structure

For validating a topological structure, first a complete set of validation rules must be defined. Based on these rules, validation tests, to check the validation rules, could be designed and implemented on the topological structure (chapter 5).
Validation can relate to both individual primitives and to the relationships between them. In this case it is about the validation of the whole structure. In order to define a valid structure each associated primitive (each single volume) must be valid, as well as the relation between all volumes. The topological structure will be validated according to a predefined set of rules. These rules are based on the discussed examples.

The topological structure consists of volumes, each volume and all its primitives it is made of, need to be valid as well. Based on the requirements and the conceptual model of the topological structure (chapter 3) and the discussed validation rules a definition of a valid structure is set. Based on this definition nine validation rules are defined.
The definition of a valid structure: 'a *topological space which is divided into a set of non-overlapping valid linear volumes without any gaps*'. Four aspects can be distinguished: a topological space, a set of non-overlapping volumes without any gaps, linear volumes and valid volumes.
The topological space is defined by 1 or more inner shells of the universal volume. These inner shells must be valid shells and are not allowed to intersect (touch is allowed).

---

geometries is needed, therefore the input geometry will be replaced by the geometry derived from the created topology. When needed the topological primitives will be adjusted to the replaced geometries and will be validated by a set of validation rules.

A set of non-overlapping volumes without any gaps means that every face is on the boundary of exactly two volumes, one on each side. Furthermore volumes are not allowed to intersect and no isolated and dangling primitives allowed.
Linear volumes can be established by planar faces and straight edges.
The definition of a valid volume is based on the definitions of valid solids (section 4.3.) and the requirements for the structure (chapter 3). Each volume must consist of one outer shell and zero or more inner shells. Each shell consist of 4 or more valid faces, which are non-overlapping, proper oriented and connected in a topological cycle. Shells are allowed to touch each other or themselves by node or edge (not by face). The geometric realization of each volume bounds a single volume'.

To meet the above definitions, the structure needs to meet the following rules. Since the topological primitive is a definition of an object located in space, it has to deal with some geometrical rules (this cannot be left out of account). Therefore some aspects within the validation could not be tested within the topological relationships and are based on geometry. All rules are explained below.

Topological validation
1. Unique primitives
2. Each primitive is part of a volume
3. Each undirected primitive is associated with two opposite directed primitives
4. Proper orientation
5. Each boundary is closed

Geometrical validation
6. Valid extent
7. The structure is linear
8. Inner boundaries must be inside outer boundaries
9. No intersections
10. Bounding single volumes/areas

1. Unique primitives.
Each node has unique xyz values (within the geometrical tolerance distance), which hold the coordinates of a node. Each geometrical realization must be unique. When each node is unique, each (unique) combination of primitives forming another primitive is unique as well. An edge consisting of the nodes N1 and N2 could not have the same geometry as an edge specified by the nodes N3 and N4, because all nodes have unique coordinates (irrespective of the direction of the primitive).

2. Each primitive is part of a volume.
Because the structure consists of only volumes and no isolated primitives are allowed, each primitive must be part of a volume. Shells and rings can be added to this rule: each node is part of an edge, each edge is part of a ring, each ring is part of a face, each face is part of a shell and each shell is part of a volume.

3. Each undirected primitive is associated with two opposite directed primitives.
Since the structure is a full space partition and no isolated and dangling primitives are allowed (rule 2 and rule 5) every primitive is associated with two opposite directed primitives. Each volume has a neighbour on both sides, this means each face is associated with two directed faces. Since each face is associated with two directed faces and each edge is part of a face (rule 2), each edge is associated with two directed edges. The same applies for nodes, since each edge is associated with two directed edges and each node is part of an edge (rule 2), each node will be associated with two directed nodes. Because each primitive is part of a volume (rule 2), this rule (3) can be summarized by: 'each undirected face is associated with two opposite directed faces'.

4. Proper orientation.
Rule 3 'Each undirected primitive is associated with two opposite directed primitives' requires every primitive has a positive and negative direction, but the topological structure does not only require an orientation but also requires a proper orientation. This means that every face part of a shell is oriented outward from the volume it bounds. In this way the interior of a volume can be distinguished from the exterior. The orientation of an inner ring must be opposite from the orientation of the outer ring.

5. Each boundary is closed.
When each boundary is closed and each primitive is part of a volume (rule 2), it means no dangling primitives (edges and faces) are present within the structure. The boundary of a volume is specified by its shells, the boundary of a face by its rings, the boundary of an edge by its nodes.
Each volume must exist of one outer shell and zero or more inner shells, whereas the inner shells are inside the outer shell. Each shell must be closed, which is established when each edge in the shell is on the boundary of two or more (even number) faces (of the same shell). Each face must exist of one outer ring and zero or more inner rings, whereas the inner rings are inside the outer ring. Each ring consists of 3 or more edges and must be closed. A closed ring is established when all edges are connected in a topological cycle. This means each node within the ring is part of two directed edges (within the ring), once as startnode and once as endnode. Each edge has a startnode and an endnode.

6. Valid extent
The extent is modelled by the universal volume, a volume without an outer shell, the extent is specified by zero or more inner shells.

7. The structure is linear
When all faces are planar and edges are straight, it means the structure is linear. When a face is planar, it means all nodes of that face are on the same plane (within a geometrical tolerance value). This topological structure only deals with straight edges, defined by a start- and endnode.

8. Inner boundaries must be inside outer boundaries
Every inner shell must be inside the outer shell of the same volume. Inner shells are not allowed to be inside other inner shells of the same volume. The same applies for inner rings. Inner rings must be inside the outer ring of the same face and are not allowed to be inside other inner rings of the same face.

9. No intersections
No intersections between and within shells and rings are allowed. No intersections between and within shells means no intersection of faces (within one shell or between different shells), but faces are allowed to touch (when a node or edge is present at the intersection or when it is about a node-in-face or edge-in-face singularity). No intersection between rings is allowed, although they are allowed to touch when a node or edge is present. Inner rings are allowed to touch the outer ring (of the same face), when a node is present. Outer rings are not allowed to touch themselves.

10. Bounding single volumes/areas
Every volume should bound a contiguous volume and every face should bound a contiguous area. The boundaries are already checked for being closed (rule 5). All primitives involved in the boundary are connected to each other (rule 2), but this does not guarantee that the interior is contiguous.

When all the above rules are met, a valid structure is defined. In table 4.1 an overview is presented of the validation rules in relation to the four aspects of the definition of a valid 3D topological structure. This classification does not separate the four aspects. All four

aspects are part of the valid structure, therefore the aspects need to be approached as one entity and not as separate aspects.

	Validation rule	Topological space	Full space partition	Linear volumes	Valid volumes
1	Unique primitives	***V***	***V***	***V***	***V***
2	Each primitive is part of a volume		***V***		
3	Each undirected primitive is associated with two opposite directed primitives		***V***		
4	Proper orientation	***V***			***V***
5	Each boundary is closed	***V***	***V***		***V***
6	Valid extent	***V***			
7	The structure is linear			***V***	
8	Inner boundaries must be inside outer boundaries	***V***			***V***
9	No intersections	***V***	***V***		***V***
10	Bounding single, contiguous volumes/areas	***V***			***V***

Table 4.1 The validation rules related to the definition of a valid 3D topological structure

For all aspects it is needed that the primitives are unique (rule 1). Other rules differ per aspect.
The *topological space* is defined by the universal volume, which should consist of one or more inner shells (rule 6). The inner shells of the universal volume need to be closed (rule 5), proper oriented (rule 4) and bound a contiguous volume (rule 10). Furthermore the inner shells are not allowed to intersect with each other or with themselves (rule 9) and are not allowed to be inside each other (rule 8).
Within a *full space partition* volumes are not allowed to intersect each other (rule 9) and no isolated (rule 2) and dangling (rule 5) primitives are allowed. Each face must be on the boundary of two different volumes (rule 3).
For *linear volumes* each face must be planar and every edge must be straight (rule 7).
For a *valid volume* rings and shells must be oriented properly (rule 4), each boundary must be closed (rule 5) and all inner boundaries must be inside the outer boundary of the same volume (rule 8). No self intersections are allowed (rule 9) and the volume should bound a contiguous volume (rule 10).

In the next chapter a topological structure will be implemented with validation tests based on these validation rules.

5. Implementation of the 3D topological structure (prototype)

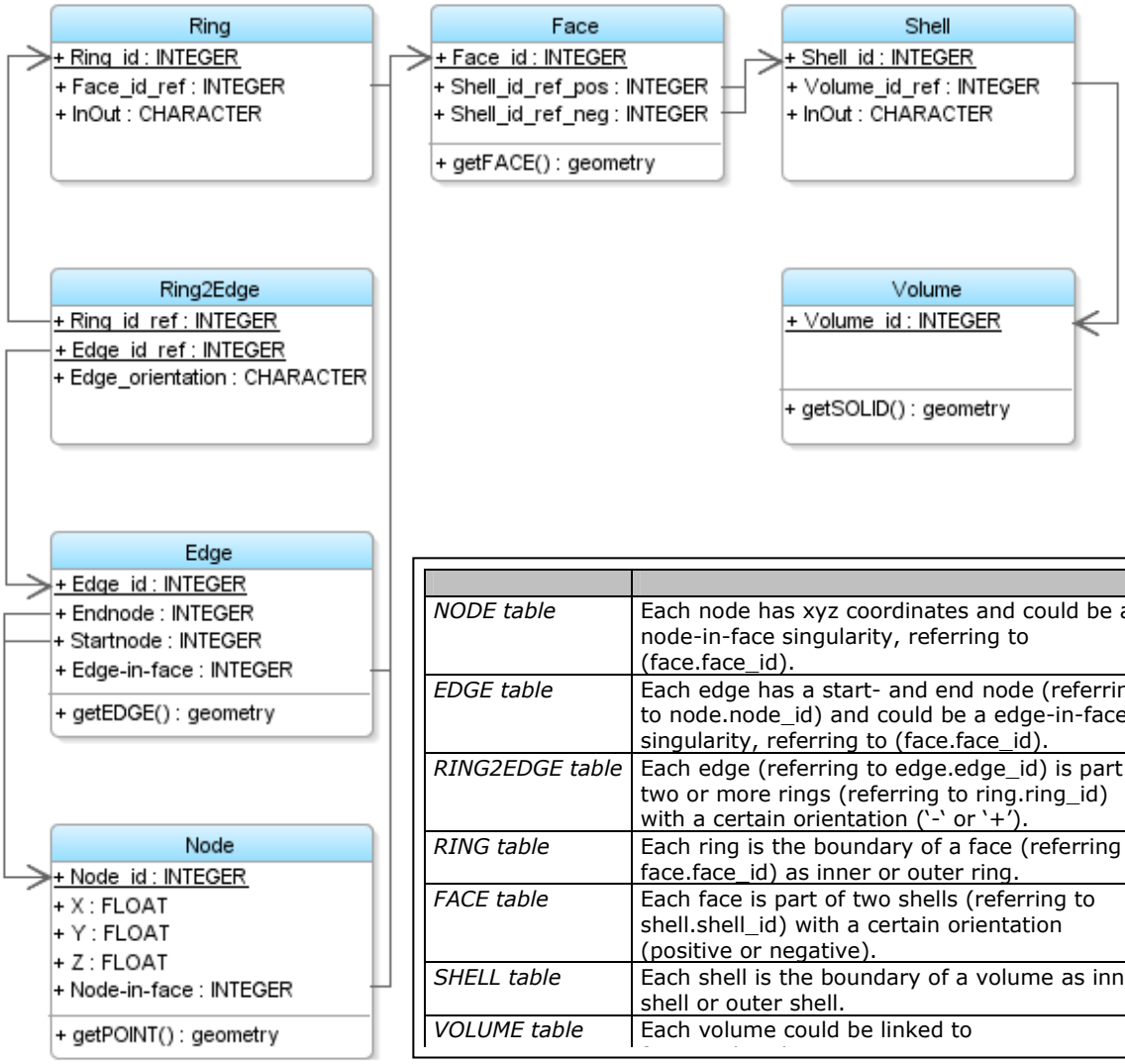
Based on the conceptual model (chapter 3) and the validation rules (chapter 4), a prototype will be implemented in Oracle in this chapter. First the logical model of the topological structure will be presented in section 5.1, followed by the technical model in section 5.2. The technical model consists of implementation specific tables with columns, data types and constraints. In section 5.3 the validation tests, based on the validation rules of chapter 4, will be designed and implemented. These tests will be illustrated with theoretical examples in section 5.4. Geometry operations will be designed and implemented in the last section (5.5). In chapter 6 the topological structure, the validation tests and the geometry operations will be tested with a data set.

5.1. Topological structure (logical model)

Different approaches exist in designing a structure. A balance has to be found between little redundancy and (relative) easy validation tests and geometry operations.[23] When a lot of redundant information is stored, the structure is large and slow and more chances for storing contradictions are present. On the other hand, it will be easier to extract information out of the structure (less derivations needed). When there is little redundant information stored, less contradictions could occur in the storage, but it will require some complex algorithms to get the needed information out of it and it will be difficult to check if the structure is still valid after an edit. The starting point for this structure will be simple tables. This means few columns and simple data types (basic data types, no collections) which will automatically lead to more tables. Another way is using less tables, but more columns and complex data types (collections). Because 3D data contains a lot of information (much more than 2D data), normalization will be a good idea. Normalization leads to less redundancy and less columns per table (but more tables). Because there are more tables and not too many columns per table, the structure will mainly make use of basic data types. It is easier and faster to extract information from basic data types and to compare data stored as basic data types. Basic data types will lead to more rows, but this will be restricted by storing relationships in an efficient way. The goal of this topological structure will be on manageable tables with little redundancy.

First the conceptual model of chapter 3 will be translated into a logical model, based on the above concepts. Then the logical model will be translated into a technical model (section 5.2).
The logical model consists of seven tables with every table having one to five columns, all having simple data types. For each primitive a table is defined, a table for rings and a table for shells is added as well as a table, linking edges and rings. For each primitive (node, edge, face and volume) the getGEOMETRY operation is added. All references are provided with primary and foreign keys. The primary key of the RING2EDGE table consists of a combination of columns, because no single column is unique. The logical model is displayed in figure 5.1.

[23] Another aspect could be efficiency (in time and storage space), but this is out of scope of this research (see chapter 1). Therefore, time and storage space will not play a crucial role within the development of this prototype.



NODE table	Each node has xyz coordinates and could be a node-in-face singularity, referring to (face.face_id).
EDGE table	Each edge has a start- and end node (referring to node.node_id) and could be a edge-in-face singularity, referring to (face.face_id).
RING2EDGE table	Each edge (referring to edge.edge_id) is part of two or more rings (referring to ring.ring_id) with a certain orientation ('-' or '+').
RING table	Each ring is the boundary of a face (referring to face.face_id) as inner or outer ring.
FACE table	Each face is part of two shells (referring to shell.shell_id) with a certain orientation (positive or negative).
SHELL table	Each shell is the boundary of a volume as inner shell or outer shell.
VOLUME table	Each volume could be linked to

Figure 5.1 The logical model

As stated in chapter 3 orientation and boundaries (rings and shells) play an important role within topology. Orientation and rings and shells (boundaries) play a crucial role within validation as well. Within the logical model orientation, rings and shells are included. With only the four primitives and the references (boundary relationships) between the primitives, it would be quite complicated to derive geometries (including inner- and outer rings/shells and orientations), especially of faces and volumes, and to validate the structure. In order to relax the validation and the derivatives (deriving the geometries of faces and volumes), orientation, shells and rings are stored explicitly. The singularities node-in-face and edge-in-face are typical topological characteristics and are stored explicitly as well.
The logical model consists of 7 tables. The VOLUME table could be related to features and attributes. A volume consists of 1 or more shells. Shells are stored in the SHELL table signed as inner (I) or outer (O) shell and related to a volume. A shell consists of a set of faces. The FACE table stores faces referring to exactly 2 shells (one positive and one negative reference). A face consists of 1 or more rings. The rings are stored in the RING table signed as inner (I) or outer (O) ring, related to a particular face. The RING2EDGE table is a link table, linking edges and rings (an edge is part of 2 or more rings). Each edge-ring combination is unique because an edge could only be used once in the boundary of a single ring. The edges with their start- and endnode are stored in the EDGE table and the nodes with their geometry are stored in the NODE table. The node-

in-face and edge-in-face singularities are stored in the NODE and EDGE table (both columns will be empty if no singularity is present). Finally geometry is stored at node level to avoid redundancy. The geometry is stored as three separate x,y,z values (simple data types). The geometry of edges, faces and volumes must be derived

Rings and shells
For the shells and rings two tables are included within the boundary relationships. Although it means two extra tables, it makes testing validity easier as well as getting geometries (of faces and volumes) and it keeps the structure transparent. By storing the rings and shells separately it could occur that an inner ring equals an outer ring of another face or an inner shell equals an outer shell of another volume. An inner ring of a face could be defined by the same edges (regardless the direction) as an outer ring of another face. This is illustrated in figure 5.2. The red ring in the left figure is the inner ring of the grey face, but also the outer ring of the yellow face. While in the right figure the red ring is the inner ring of the grey face but not the outer ring of one of the other faces.

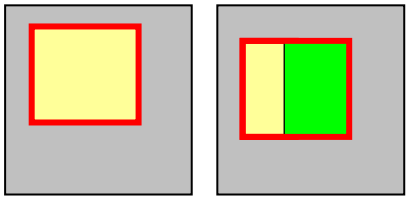


Figure 5.2 An inner ring equalling an outer ring

When an inner ring equals an outer ring (regardless their directions), both rings will be stored as separate rings (with separate id's). In favour of testing validity and defining a transparent model. The same applies to inner shells.

Inner shells are also used for modelling the universe. The universal volume will be stored with id 0 (Volume_id=0) and consists of only inner shells. The universal volume could have more than 1 inner shell, so the universe could consist of more parts. Due to modelling the universe as an inner shell, it is not possible for a universe to have voids (like an inner shell in a normal volume). The solution will be to model the void as a volume.

Orientation
Orientation is added at the edge-ring relationship. The orientation of edges is stored by references to a start- and endnodes. The orientation of rings will be stored as a random direction. In 3D it does not matter which direction, only inner rings must be oriented in opposite way to the outer rings. This is established by storing the orientation ('+' or '-') of each edge within a ring at the edge-ring relationship in the RING2EDGE table. The orientation of a face is determined by the orientation of the ring(s). Finally the orientation of a face within a shell (face-shell relationship), is stored in addition to the (random) orientation of each single face. The orientation of faces within shells is stored at the face-shell relationship in the FACE table. A face has exactly two shell references, one negative and one positive reference, for both references a separate column within the table is admitted.

What if no orientation was stored…
What if no orientation was stored and a valid volume geometry was needed with faces pointing outward. What if the edges are stored without orientation and are just bounded by two (unordered) nodes (no start- and endnode). Then an edge is randomly used in a ring (without a positive or negative direction) and a ring could be composed by connecting all edges by their nodes. The ring will have a random direction. For distracting polygons (without holes) in 3D this method can be used (because a 3D polygon does not has a predefined direction in 3D), but when distracting a solid it will be very complicated. For every face within the volume the direction needs to be correct. First the direction of a face has to be defined (the normal pointing outward for a face in an outer shell), then the face has to be constructed in this particular direction, this process is needed for every face in the volume. Next to the operation 'get geometry of volume', orientation of faces is also needed for several validation tests, like the closed volume test.

With the storage of orientation, checking for validity is much easier as well as deriving the geometries of volumes. A disadvantage is the extra storage and the chance of storing contradictions; more validity tests are needed, but this is no proportion to the advantages.
In this model (the logical model, figure 5.1) a balance has been found between simplicity and usability. The structure consists of seven tables with every table having one to five columns, with all simple data types. Next, this logical model will be translated into a technical model with implementation specific details.

5.2. The implementation (technical model)

The implementation is done within Oracle Spatial, therefore Oracle proprietary data types are used. For the geometry operations, the output will be in SDO_GEOMETRY. Another aspect influencing the (volume) geometry operation is that Oracle's solid (SDO_GEOMETRY) does not allow inner rings. Since the topological structure does allow inner rings, a solution is needed when realizing a geometrical solid from a topological volume with an inner ring. In such cases, a face needs to be split in such a way that no inner rings exist anymore. For reasons of simplicity, this prototype is restricted to only one inner ring per face. In theory there is no restriction on dissolving inner rings. An edge needs to be added at the shortest distance between two rings till all inner rings are 'dissolved'.
Compared to the logical model, the technical model displays implementation specific tables. This means that the data types of the technical model are refined into Oracle data types. The operations are not displayed in the technical model, because they are not stored within the tables (the operations are discussed in section 5.5). Furthermore, two views are added: NODE2SHELL and RINGORDINATES.
The disadvantage of using more tables (in this case seven) is the 'long way' between the VOLUME table and the NODE table. Therefore a view (NODE2SHELL) is added, which bridges between the NODE table (first in row) and the SHELL table (sixth in row). The view is derived from the tables EDGE, RING2EDGE, RING, FACE and SHELL and presents all node-edge-ring-face-shell combinations without any orientation information. Despite the view NODE2SHELL the 'long way' must still be travelled when needing orientation (the view does not hold orientation information). This is at least the case when realizing solids or polygons. Therefore the view RINGORDINATES is added. This view displays the ring-ordinates (SDO_ORDINATE_ARRAY) of each ring, providing extra geometry information in addition to the geometry stored at node level.[24] Unfortunately Oracle does not support collection data types in a virtual column, therefore the view was added instead of an extra column in the RING table. Next to the two geometry operations (getPOLYGON and getSOLID), the ring-ordinates are used in some validation tests as well. From an ordinate-array it is easy to extract node-ring information and it is easy to form geometries from. The technical model is displayed in figure 5.3.

[24] The column RINGORDINATES is populated with the help of the function getRINGordinates (see appendix I).

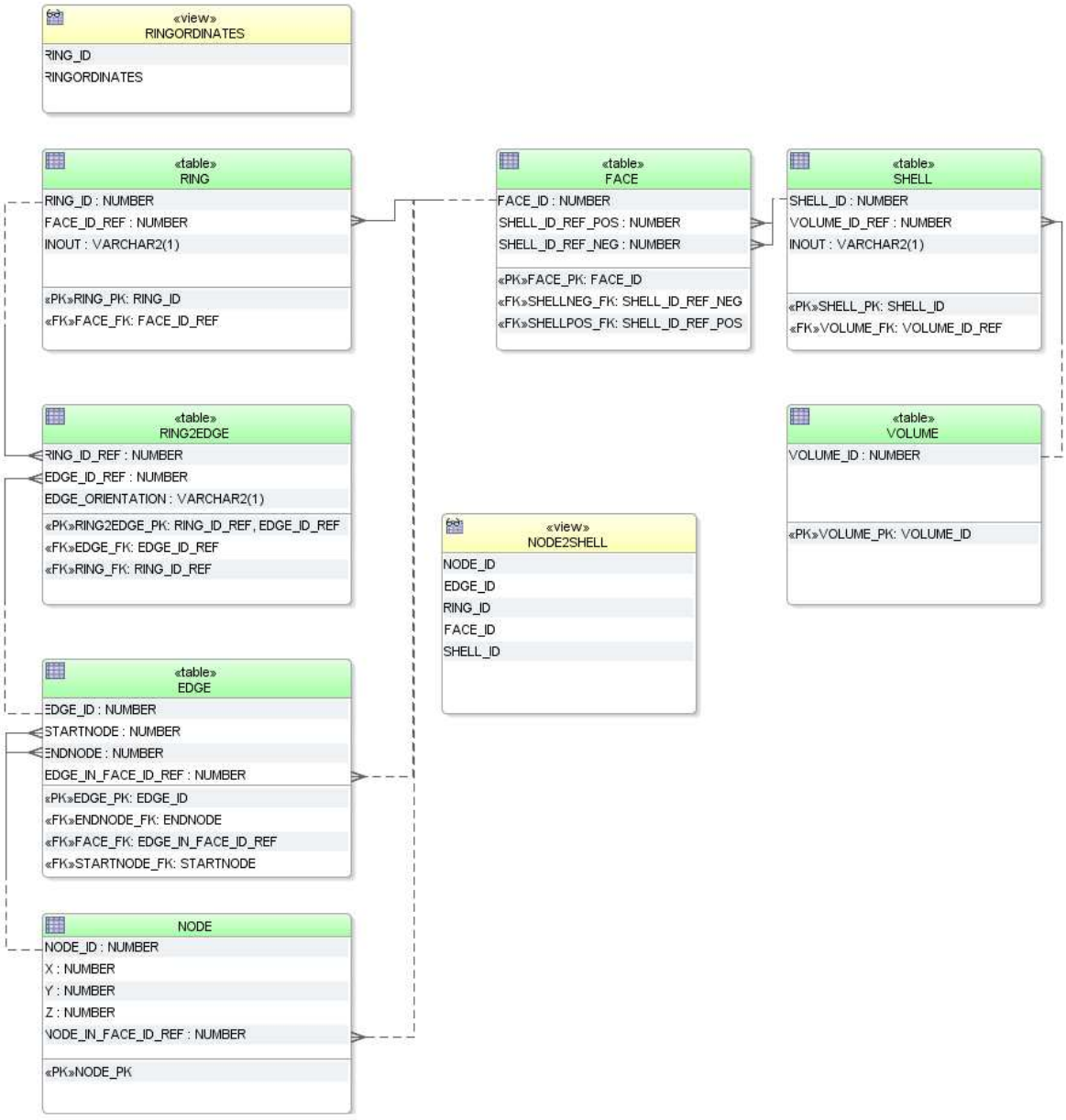


Figure 5.3 The technical model

In the next section, validation tests based on the validation rules of chapter 4 will be designed and implemented on the above model.

5.3. Validation tests

A topological structure has its strengths (and weaknesses) compared to a geometrical model. This means that validation tests based on geometrical characteristics, which is inevitable, will be reduced to a minimum and the validation tests will be based, as much as possible, on the topological relationships between the nodes, edges, faces and volumes.

Simple (fast) tests will be done on the whole structure and reduce the hard (time consuming) tests as much as possible. It is better to have more simple tests than to having less but harder tests. Some constraints are enforced by the use of primary and foreign keys and non nullable columns. However not too many unique keys are applied because this will make the structure very inflexible (they could better be replaced by simple tests). When the structure is populated the keys could be disabled and after

finishing enabled again. The same could be done when editing, but it is easier to make use of the DBMSs commit function.
When the structure is populated or edited, validation is needed. A global validation for the whole structure after population and local validation after editing. The validation tests are divided into three sets of validation tests: the primitive block, the orientation block and the geometry block.
First some basic (simple) tests need to be done on the primitives, those tests are enclosed in the primitive block. Then, some tests need to be done related to orientation, which are enclosed in the orientation block. Finally, some tests need to be done based on geometrical characteristics, which are enclosed in the geometry block. It is inevitable to perform some geometry related tests. For example, the structure cannot do without the rule 'volumes are not allowed to intersect'. By applying several topological tests ('primitives block' and 'orientation block') volumes need only to be tested on self-intersections. In this way a complicated test is reduced compared to testing also volumes intersecting each other (see test 7). It is also important to keep the tests efficient and gear the tests to one another, so things are only checked once.

Tolerance values play an important role within validation. First of all at defining unique nodes; depending on the scale of the structure a tolerance value should be set, in order to detect nodes which could be assumed to be the same. Tolerance values are out of scope of this research. All geometrical related tests, like the test 'no intersections' (test 7), have to deal with tolerance values. Tolerance values are not specifically developed for this test in this stage, but the test makes use of an existing Oracle function and therefore also of the accompanying tolerance options. Test 6 'linear structure' which tests for planar faces has a tolerance value build in.

Based on these principles, the ten validation rules of chapter 4 have been translated into 8 validation tests. This next paragraph will give a description on how the ten validation rules are translated to tests based on the topological structure. First an overview of the constraints, generated by primary and foreign keys and not allowing NULL values, and their implications, is given in table 5.1. These constraints have been applied to the id's of the primitives and their references, which does not automatically mean that the primitives themselves are unique (this will be tested in the validation tests).

Implication	**Constraint**
Nodes:	
Each node (admitted in the NODE table) has an unique id	Primary key
Each node has xyz coordinates	Not nullable
Each node related to a face (node-in-face) refers to an existing face	Foreign key
Edges:	
Each edge (admitted in the EDGE table) has an unique id	Primary key
Each edge has a start- and endnode	Not nullable
Each edge (admitted in the EDGE table) refers to existing nodes	Foreign key
Each edge related to a face (edge-in-face) refers to an existing face	Foreign key
Each edge (admitted in the RING2EDGE table) refers to an existing edge	Foreign key
Each edge (admitted in the RING2EDGE table) is part of a ring	Not nullable
Each edge (admitted in the RING2EDGE table) refers only once to the same ring	Primary Key
Rings:	
Each ring (admitted in the RING table) has an unique id	Primary key
Each ring (admitted in the RING2EDGE table) refers to an existing ring	Foreign key
Each ring (admitted in the RING table) is part of one face (no more no less)	Not nullable
Faces:	
Each face (admitted in the FACE table) has an unique id	Primary key
Each face (admitted in the RING table) refers to an existing face	Foreign key

Each face (admitted in the FACE table) is part of exactly two shells (no more no less) in two different directions	Not nullable
Shells:	
Each shell (admitted in the SHELL table) has an unique id	Primary key
Each shell (admitted in the SHELL table) is part of one volume (no more no less)	Not nullable
Each shell (admitted in the FACE table) refers to an existing shell	Foreign key
Volumes:	
Each volume (admitted in the VOLUME table) has an unique id	Primary key
Each volume (admitted in the SHELL table) refers to an existing volume	Foreign key

Table 5.1 The implications of the constraints of the tables

The relationship between the eight validation tests and the ten validation rules is presented in table 5.2. All tests will be provided with a short explanation, the actual PL/SQL-scripts of the tests are added in appendix I. In this stage all references in the scripts will be 'hard' references, no dynamic SQL is used.

The validation tests are divided into three blocks, based on their main characteristics: the primitives block, the orientation block and the geometry block.

Rule	Test	Block
1. Unique primitives	1	Primitives
2. Each primitive is part of a volume	2	Primitives
3. Each undirected primitive is associated with two opposite directed primitives	n.a.	n.a.
4. Proper orientation	5/4	Orientation
5. Each boundary is closed	3/4	Orientation
6. Valid extend	3	Orientation
7. The structure is linear	6	Geometry
8. Inner boundaries must be inside outer boundaries	7	Geometry
9. No intersections	7	Geometry
10. Bounding single volumes/areas	2/8	Geometry

Table 5.2 Validation rules linked to the validation tests

As table 5.2 shows rule 3 'each undirected primitive is associated with two opposite directed primitives' has no associated validation test. As explained in chapter 4 (section 4.2 - rule 3) for a full space partition only faces need to be checked for being associated with two directed faces (the direction is passed through to the other primitives). Every face has exactly two shell references; 1 negative and 1 positive, which is exacted by the columns FACE.SHELL_ID_REF_POS and FACE.SHELL_ID_REF_NEG (and the primary key on the face_id). Therefore no test is needed for testing 'each face is associated with two (opposite) directed faces'.

'Primitives block'

The primitive block consists of two tests, testing the uniqueness and references of the involved primitives. When the structure has passed these tests, tests of the next block could be started.

5.3.1. Test 1) unique primitives

For this test 3 subtests are performed for unique nodes, edges and faces. Unique volumes do not need to be tested because volumes can not refer to the same shell (by the primary-key 'shell_PK' on the shell in the SHELL table) and all shells are unique (if all faces are unique).[25]

[25] Shells (with different id's) could not consist of the same faces. Every face has exact two shell references: 1 negative and 1 positive (exacted by the columns FACE.SHELL_ID_REF_POS and FACE.SHELL_ID_REF_NEG and

Test 1a) unique nodes
All nodes already have unique id's (by the primary key on NODE.NODE_ID), but still need to be checked on unique xyz coordinates. Since no tolerance value is added (out of scope of this research), nodes within the tolerance value will be tested FALSE.
Test 1b) unique edges
All edges already have unique id's (by the primary key on EDGE.EDGE_ID), but still need to be checked on unique references to their two nodes. Within the EDGE table, a startnode and endnode of an edge may never be the same. The combination startnode-endnode must be unique (no two edges may refer to the same combination). And a startnode-endnode combination may not equal another endnode-startnode combination, because the opposite direction of the edge is stored at the ring (by the use of a '-' sign) in the RING2EDGE table.
Test 1c) unique faces
All faces already have unique id's (by the primary key on FACE.FACE_ID), but still need to be checked on unique references to their edges. This means that each face must have a unique reference to (the combination of) rings and edges. Faces can not refer to the same ring (by the primary-key 'ring_PK' on the ring in the RING table), although rings (with different id's) could consist of the same edges and actually be the same ring. An exception is an inner ring being the same as another outer ring (different id's, but same references to edges), this is allowed and will be tested valid. The test could be slimmed down to testing for equal outer rings (only[26]). This is done by checking the collection of edges per outer ring (irrespective of the order).[27]

5.3.2. Test 2) primitive references

In this test the references between the primitives (one dimension higher/lower) will be tested in 3 subtests; node-edge, edge-face and face-volume, in order to avoid isolated (no reference to a primitive in a higher dimension) and nonexistent (no reference to a primitive in a lower dimension) primitives. This check does not exclude dangling edges and faces, which will be checked in test 4. Checking the references is partly taken care of by not allowing NULL values in the tables and applying foreign keys, but it is necessary to perform some additional tests for isolated nodes and edges and for nonexistent faces, rings and shells. A volume can not be isolated, because the volume is the highest dimension in the structure. A node can not be nonexistent because the node is the lowest dimension.
The primitives are also tested for their minimum number of boundary primitives: a volume must consist of 4 or more faces and a face must consist of 3 or more edges. An edge must consist of exactly two nodes, but this is taken care of by the columns startnode and endnode in the EDGE table and therefore does not need to be tested.
The third aspect in this test deals with primitives used more than two times in one boundary. A node could be on the boundary of many edges, but an edge can only be used once in the boundary of a face (irrespectively of being an inner or outer ring) and a face can only be used once on the boundary of volume.

Test 2a) node-edge references
All nodes are tested for isolation, they must be admitted in the EDGE table as start- or endnode.

---

the primary key 'face_PK') in addition every face is unique (which is tested). Even when two shells have the same faces refer to it, the orientation will always be in opposite direction. This could happen (and is allowed) when an inner shell equals an outer shell (different shell-id's but consisting of the same faces, although oriented in opposite direction).
[26] Assumption: It is not possible to have faces refer to the same outer ring and still be unique and valid.
[27] Assumption: It is not possible to have unique and valid faces with the same collection of edges, regardless the order of the edges.

Test 2b) edge-face references
All edges are tested for isolation and all rings for nonexistence. By the foreign keys on edges and rings ('edge_FK' and 'ring_FK') in the RING2EDGE table, nonexistent edges and isolated rings are not possible.
Furthermore, each ring will be tested for 3 or more (unique) edges and each face (with an inner ring) will be tested for edges used more than 1 time in a face. An edge can not refer more than one time to the same ring (by the primary key 'RING2EDGE_PK'), but inner and outer rings (of one face) could refer to the same edge (which is not allowed). Finally, within a ring, a node may only be bounding 2 edges, but it is allowed for one node (only) to be in the inner ring and outer ring of the same face. More than one node will lead to a non-contiguous area, which is not allowed.
Test 2c) face-volume references
All faces and shells are tested for nonexistence. By the foreign keys on the face and shell references ('face_fk', 'shellpos_fk' and 'shellneg_fk') isolated faces and shells are not possible. Furthermore, each shell in a volume is tested for consisting of four or more (unique) faces. Finally, each volume is tested for unique faces. A face may only occur once in a volume (including inner and outer shells). This means that a face can not refer twice to the same shell (one neg. and one pos. reference). It is also not allowed for faces to refer to the inner and to the outer shell of one volume.

'Orientation block'
Before testing the orientation, the structure has to pass the above tests of the primitive block first.

5.3.3. Test 3) each face/volume consists of one outer boundary

Each face is tested for exactly 1 outer ring (no more and no less than 1 outer ring) and each volume for 1 outer shell (two subtests). The universal volume is an exception and consists of only inner shells.[28] In this structure (at the moment) only one inner ring is allowed, but in theory much more inner rings could occur without any problems.[29]

Test 3a) each face consists of one outer ring
Each face is tested for exactly 1 outer ring (no more and no less than 1 outer ring). No faces with only inner rings are allowed and no faces with more than 1 outer ring are allowed. For the moment a test is added for a maximum of 1 inner ring.
Test 3b) each volume consists of one outer shell
Each volume is tested for exactly 1 outer shell (no more and no less than 1 outer shell). The universal volume is an exception and exists of only one or more inner shells. No volumes with only inner shells are allowed and no volumes with more than 1 outer shell are allowed. The universal volume is a special volume, which needs to be tested on its (correct) presence. The universal volume (volume_id=0) must have at least one inner shell and no outer shells.

5.3.4. Test 4) closed boundaries

Each boundary has to be closed. The boundary of an edge is closed when bounded by two unique nodes, this is already secured by test 1. The boundaries of faces and volumes are checked by testing for closed rings and shells (by the orientation of the edges and faces within the rings and shells).

Test 4a) closed ring
A face is closed when each ring in the face is closed. The function getRINGordinates takes care of the test for a closed ring.[30] The view RINGORDINATES is created with this function. This function establishes a string of ordered nodes and tests if the ring is closed

[28] A universe with a void (disconnected universe) could not be modelled at this stage. The solution is modelling the void as a normal volume (see section 5.1).
[29] See section 5.2.
[30] Every so called assistant function, like getRINGORDINATES, is in Appendix I, including a summary.

(last node equals the first node) and checks the orientation (RING2EDGE.ORIENTATION) of each edge (in a ring). When the ring does not meet these requirements, the functions returns a NULL value.
Test 4b) closed shell
A volume is closed when each shell in the volume is closed. A closed shell is established when each edge in the shell is on the boundary of two or more (even number) faces (of the same shell). This means that each edge, involved in the shell, must have an equal distribution of negative and positive references. When checking the distribution of negative and positive edge references in each shell, consideration is needed for the orientation of the faces in a shell next to the orientation of the edges in a face.

5.3.5. Test 5) proper orientation

Every face in a shell is oriented outward (positive). For the orientation of rings within a face, it is slightly different, because it is not possible to define a (counter)clockwise or inward/outward orientation for a single face in 3D. The only restriction on the orientation of faces is that the orientation of an inner ring must be opposite from the orientation of the outer ring (of the same face). Therefore, only inner rings will be tested for their orientation.
Because the structure defines a full space partition, without dangling (test 4) and isolated primitives (test 1) and with all primitives associated with two directed primitives (rule 3), the orientation of all primitives within each single 'universe' is geared to one another. With 'universe' all inner shells are meant. The total universe is defined by the universal (inner) shell, but within this universe there are 'mini universes' defined by other inner shells. This means that the orientation of each inner shell (including the universal shell) must be checked for proper orientation.

Test 5a) each inner ring must have a proper orientation
The normal of a plane is perpendicular to the plane and is calculated as the cross product of two (non-parallel) vectors, lying on the plane. When the normal of the inner ring is opposite to the normal of the outer ring, the orientation is correct.[31] In this test, a tolerance value (0.01) is added at the comparison of the normal values, which could be adjusted per data set.
Test 5b) each inner shell must have a proper orientation
It is enough to test the orientation of just one face of the inner shell, because all orientations are geared together in a closed boundary by the equal distribution of edges (checked by test 4). Therefore one face of each inner shell is tested for a proper orientation (pointing outward the volume) with the help of the function getNORMAL.[32] When a face with the lowest z-value, which is not vertical (along the y-axis), has a negative z-value for its normal, the face, and thus the inner shell, is oriented properly.

'Geometry block'
This final geometry block will test the structure on some geometrical related validation rules.

5.3.6. Test 6) linear structure

When all faces are planar the structure will be linear. A face is planar when all nodes of that face are on the same plane (within a planarity tolerance value). In this test, faces will be tested for their planarity.

Test 6a) planar faces
Each ring (inner and outer rings) of a face will be tested on planarity by calculating the distance from each node to the plane (a tolerance value is added).[33] The plane is defined

[31] When a face in not planar (test 6a) the face will not pass this test (test 5a) either.
[32] Every so called assistant function, like getNORMAL, is in Appendix I, including a summary.
[33] At the moment the planarity is tested by calculating the distance of the rings centroid (the mean of all x, y and z values) to the plane of the ring. The centroid will averaging out the distance of the nodes to the plane

with the help of two random non-parallel edges. Next, all inner rings will be tested on being on the same plane as their accompanying outer ring by calculating the distance of the centroid of the inner ring to the plane of the outer ring (test for 'sunken' inner rings). A tolerance value (within the distance node-plane) is admitted in this test and could be adjusted depending on the data set.

5.3.7. Test 7) no intersections

Primitives are not allowed to intersect but are allowed to touch. This can be summarized in 'volumes are not allowed to intersect and self-intersect (but touch)', which can be translated into 'faces are not allowed to intersect and self-intersect (but touch)'. When all volumes are not self-intersecting, no intersecting volumes will be present because the structure is a full space partition (and each face is part of exactly two different shells). Therefore, volumes do not need to be tested on intersections between each other but each volume has to be tested on self-intersection. This will be done by testing each volume for intersecting faces. In four particular cases touching faces are valid; at an edge-sharing or node-sharing touch and at an edge-in-face or a node-in-face singularity. Self-intersecting faces will always lead to a face intersection somewhere in the volume, therefore no separate test for self-intersecting faces is needed. Inner rings are allowed to touch their outer rings (in one node only, which is tested in test 2b).
In addition, inner rings and inner shells need to be tested for not being completely outside the accompanying outer ring or outer shell (except for the universal volume) and outer rings and outer shells need to be tested for not being completely covered by their accompanying inner ring or inner shell.

Test 7a) no self-intersecting volumes
Faces are not allowed to intersect other faces of the same volume, except for touching under certain circumstances (edge-sharing touch, node-sharing touch, edge-in-face or node-in-face). Faces of one volume are allowed to touch when there is a node or edge present which is part of both faces, or when a node-in-face or edge-in-face singularity happens. This test tests at first for (an unspecified) interaction between the faces, with the help of Oracle's SDO_ANYINTERACT (which is fitted with a tolerance value). When an interaction is detected (which is not an edge sharing interaction), then the faces are tested for a node sharing touch or node-in-face or edge-in-face singularities.
Test 7b) every inner ring must be inside the accompanying outer ring
Even though inner rings and outer rings do not intersect (test 7a) and are on the same plane (test 6a), the inner ring could still be completely outside the outer ring, or the outer ring could be covered by the inner ring. Both cases will be detected when testing all edges of the inner ring being inside the outer ring. This is done with the help of Oracle's SDO_ANYINTERACT.
Test 7c) every inner shell must be inside the accompanying outer shell
The same applies for the inner shells; every inner shell must be completely inside the accompanying outer shell (but is allowed to touch), except for the universal volume, which does not have an outer shell. Each inner shell (except for the inner shells of the universal volume) will be tested for being completely inside the outer shell. This is done with the help of Oracle's SDO_ANYINTERACT. Because the inner shells of the universal volume do not have an outer shell, they must be tested for not intersecting each other and not being inside each other.[34]

5.3.8. Test 8) bounding single volumes/areas

Each volume and each face must bound a single, contiguous volume or area. Faces are already tested for self-intersections (test 7a) and for nodes used more than 2 times (test 2b). When tests 1 till 7 are passed, a face represents a contiguous area.

---

and is therefore not suitable for this test. The test should be adjusted by calculating the distance of all nodes to the plane, as described above.
[34] Tests for inner shells of universal volume must still be added.

Most volumes will be bounding a single volume when tests 1 till 7 have been passed. However, some situations could lead to breaking the rule 'bounding single, contiguous volumes'. This is the case when edges are used more than 2 times in one volume (edges are on the boundary of more than 2 faces) or when a volume has inner shells. The occurrence of the singularities node-in-face and edge-in-face will not endanger a contiguous volume, since those singularities never split a face completely (see chapter 3, figure 3.10).

Test 8a) single volumes
Some very complicated situations hamper this test, see figure 5.4 for an example. No 3D topology structure based solution has been thought about. A solution for this problem is suggested by Kazar, Kothuri, van Oosterom & Ravada (2008). They suggest the tetrahedronisation of a volume. Next, all connected tetrahedra will be marked via shared triangles (by a Boolean value), starting at a random tetrahedron. When 'travelled' all connected tetrahedra, the number of marked tetrahedra must be equal to the total number of tetrahedra. In that case a volume is contiguous. Otherwise the volume is not. This test has not been elaborated any further in this research.

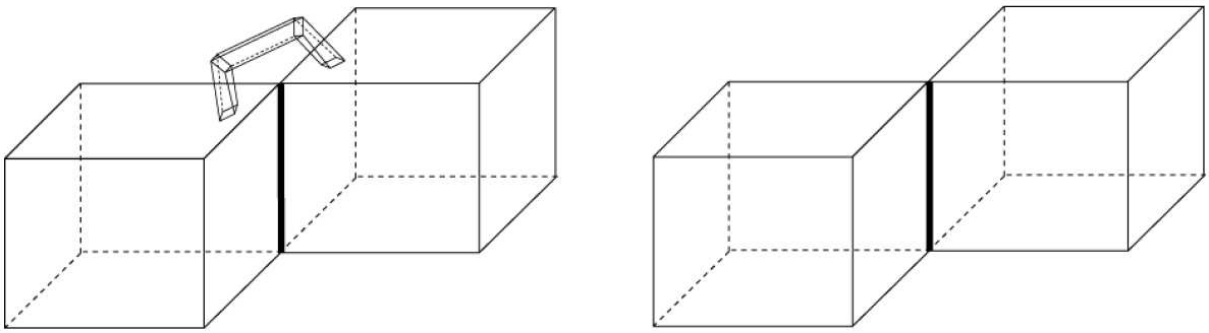


Figure 5.4 Left a contiguous volume; right a non-contiguous volume
(taken from: Kazar, Kothuri, van Oosterom & Ravada 2008)

All PL/SQL scripts are added in the appendix I, including four assistant functions which are used in some tests. All tests are written in static SQL (no dynamic SQL), therefore the scripts for creating the structure are added in the same appendix as well. At the moment the tests run on the whole dataset (global testing), but internally they test each object separately therefore it could be very useful for a local test as well (for example after an edit/update).

5.4. Illustrations of validation

In order to visualize the validation tests some illustrations will be discussed. The valid and invalid 3D geometries as pointed out in the article 'on valid and invalid three-dimensional geometries' by Kazar, Kothuri, van Oosterom & Ravada (2008) will be subjected to the above validation tests as were they modelled as topological volumes within a full space partition. These illustrations are all theoretical cases and not tested for real. A test case with real test data will be performed later, in chapter 6.

Illustration: Inner rings
Inner rings within faces are allowed in the topological structure, even when they are on the boundary of a volume. This in contrary to the geometrical solids in Oracle, where, inner rings are not allowed in polygons on the boundary of a solid. Therefore, the volume on the right (figure 5.5) should be tested valid.

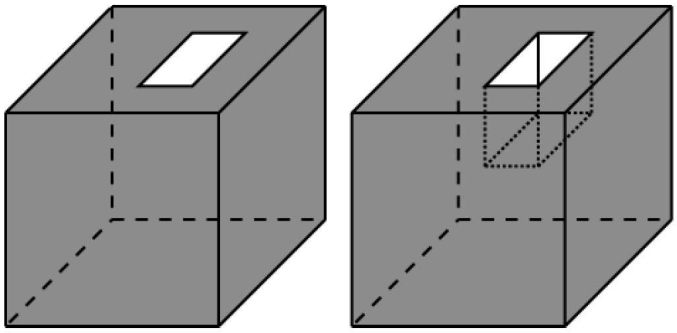


Figure 5.5 A volume with an inner ring
(taken from: Kazar, Kothuri, van Oosterom & Ravada 2008)

The left volume should be tested invalid when the white area is not a face; the boundary of the volume is not closed (test 4b). When the white area is modelled as a face the volume should be tested valid.

Illustration: Inner shell touches outer shell (by face).
Inner shells are allowed within the structure and are allowed to touch the outer shell, but only with an edge or node. An inner ring is not allowed to touch an outer ring with by face.

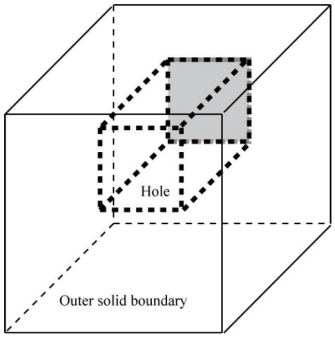


Figure 5.6 An inner ring touching an outer ring by face
(taken from: Kazar, Kothuri, van Oosterom & Ravada 2008)

The situation in figure 5.6 should be tested invalid when the grey area is defined as a separate face in the outer shell. Because this particular face will be used twice in one volume (test 2c 'face-volume references'). When the grey area is not defined as a separate face in the outer shell the situation will be tested invalid as well. At test 7a 'no self-intersecting volumes' an invalid touch should be detected.

Illustration: One volume inside another volume.
A volume with one outer shell cannot be inside another volume with one outer shell.

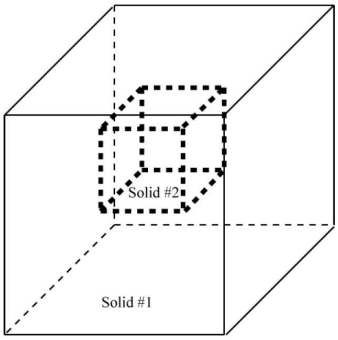


Figure 5.7 A volume inside another volume
(taken from: Kazar, Kothuri, van Oosterom & Ravada 2008)

For this situation one has to keep in mind that the volumes are modelled in a full space partition. That means that each face is always on the boundary of exactly two volumes (shells), which is provoked by the columns in the FACE table. When there are only two volumes (with only outer rings) in the structure, it means every face must be on the boundary of volume 1 and 2, on the boundary of volume 1 and 0 (universal volume) or on the boundary of volume 2 and 0. Since volumes 1 and 2 do not share a face, it is assumed that all faces of these two volumes are on the boundary of one of the two volumes and on the boundary of the universal volume. This actually creates two mini universes (two inner shells of the universal volume), which are both closed, but are intersecting with each other. This intersection should be tested invalid (test 7c). When only one universe was created (for both volumes), the 'closed boundaries' (4b) should have failed for the inner shell of the universal volume and have tested invalid. If both volumes are part of a larger structure, with more volumes and with all volumes connected to each other, and closed, at a certain point a volume will be tested invalid for self-intersections (test 7a), see also the next illustration.

Illustration: Self-intersection.
It is not allowed for a volume to self-intersect (see the right volume, figure 5.8). However it is allowed for a volume to have a node-in-face singularity, like the left volume.

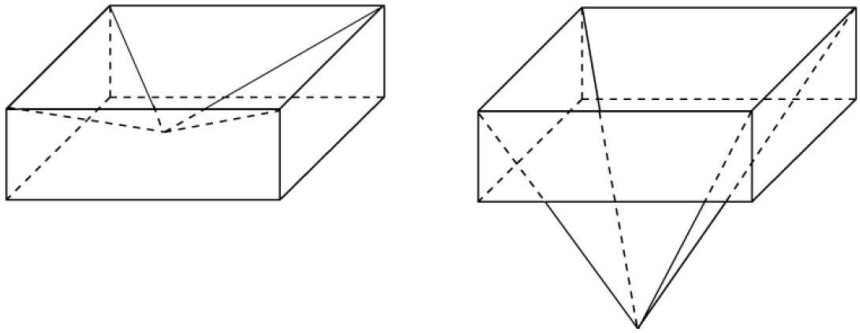


Figure 5.8 A self-intersecting volume
(taken from: Kazar, Kothuri, van Oosterom & Ravada 2008)

Assuming the left volume presents a node-in-face singularity, both situations should first be tested for the presence of an unspecified interaction (test 7a). Finally the left volume should be tested valid (as a node-in-face singularity) and the right volume should be tested invalid as an invalid self-intersection.

Illustration: Complex outer shell (edges used more than 2 times).
Outer shells are allowed to touch themselves by using an edge more than 2 times (as long as it is an even number).

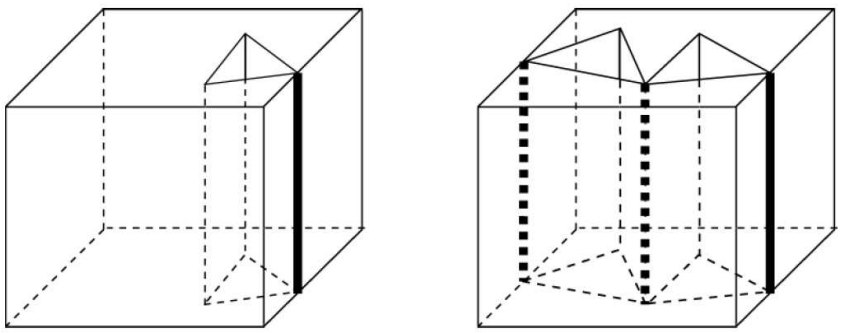


Figure 5.9 A complex outer shell
(taken from: Kazar, Kothuri, van Oosterom & Ravada 2008)

When an outer shell touches itself (creating a tunnel), like the left volume in figure 5.9, the situation should be tested valid when the top and bottom faces are modelled as an outer ring with an inner ring. Because the volume still bounds a contiguous volume. It could also occur that more than one edge is used more than two times, two 'tunnels' are created (right volume figure 5.9). This situation should be tested invalid because the volume does not bound a contiguous volume (test 8a).

Illustration: Inner shell touches outer shell (by edge-in-face).
An inner shell is allowed to touch an outer shell by an edge-in-face. In those cases the particular edge is modelled as edge-in-face singularity. When the edge-in-face singularities are valid (not splitting a face in two parts) the situation should be tested valid (if all other validation criteria are met). Both volumes in figure 5.10 are valid, as long as the top face of the right volume is modelled as two separate faces. When this top face is modelled as one face, it should be tested invalid because the face does not bound a single area (test 2b).

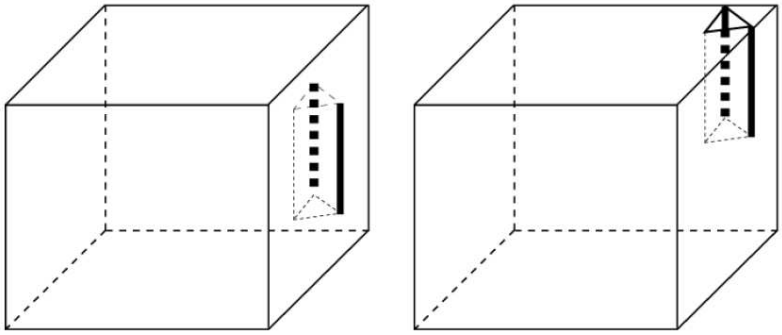


Figure 5.10 Inner shells touching the outer shell by edge-in-face
(taken from: Kazar, Kothuri, van Oosterom & Ravada 2008)

Illustration: Connected by 'handle'.
A situation like in figure 5.11 should be tested invalid, if there would not be a handle. The volumes would not be contiguous. By the adding handles, both volumes become valid and should be tested valid (test 8a single volume).

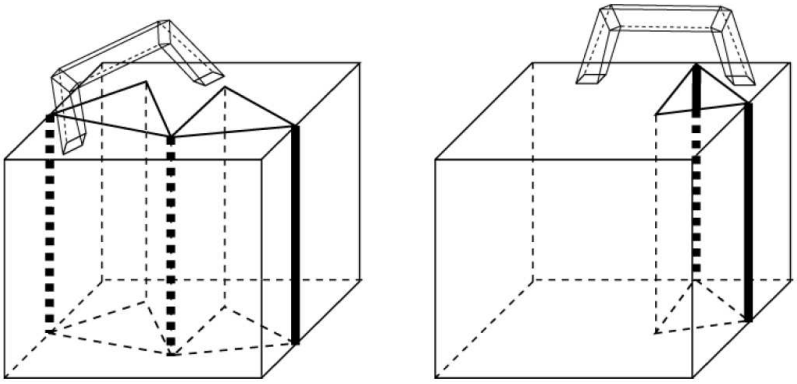


Figure 5.11 Connected by 'handle
(taken from: Kazar, Kothuri, van Oosterom & Ravada 2008)

These were some illustrations based on theory. In chapter 6 the structure and the validation method will be tested with a real data set, but first the implementation of the geometry operations will be discussed.

5.5. Geometry operations

For each primitive a geometry operation is implemented on the topological structure. Since the topological structure will be implemented within Oracle Spatial, these operations have to deal with the SDO_GEOMETRY data type. Therefore a small

elaboration on Oracle's single geometry structure (SDO_GEOMETRY) will follow. The technical specifications are presented in the following frame (Murray et al. 2008a).

Specifications of the SDO_GEOMETRY data type

1. SDO_GTYPE (NUMBER)
2. SDO_SRID (NUMBER)
3. SDO_POINT (SDO_POINT_TYPE)
4. SDO_ELEM_INFO (SDO_ELEM_INFO_ARRAY)
5. SDO_ORDINATE (SDO_ORDINATE_ARRAY)

1. SDO_GTYPE

The value is 4 digits and consists of:

- Digit 1: the number of dimensions.
- Digit 2: linear referencing system (LRS) geometry, when n/a specify 0.
- Digit 3+4: according a list.

Compulsory

2. SDO_SRID

Coordinate system, can be NULL

Not compulsory

3. SDO_POINT

Can be used to store point geometries (then SDO_ELEM_INFO and SDO_ORDINATES are NULL), other wise it is NULL.

Not compulsory

4. SDO_ELEM_INFO

The value is a triplet:

- Value 1 (SDO_STARTING_OFFSET): offset within the SDO_ORDINATES (the first triplet always starts at 1; when concerning compound objects, for each triplet the start off is given)
- Value 2 (SDO_ETYPE): type of element, according a list (related to the GTYPE)
- Value 3 (SDO_INTERPRETATION): when concerning a compound geometry, it names the number of elements, when it is not concerning a compound object, it describes a subtype (according a list).

Compulsory (can be NULL when concerning point geometries)

5. SDO_ORDINATE

All ordinates (clockwise for inner boundaries and anti-clockwise for outer boundaries)

Compulsory (can be NULL when concerning point geometries)

Different 3D primitives can be stored within SDO_GEOMETRY; points, linestrings, polygons (called surface), surfaces (called composite surface), simple solids and composite solids. All these data types have also a collection variant. In this research four single, no collection) geometry types will be used (in the getGEOMETRY operations); points, linestrings, polygons and solids. The specific characteristics of each data type is summarized in table 5.3 (for more information see Kothuri, Godfrind & Beinat 2007), the validation criteria were already discussed in chapter 4.

	Solid	Polygon	Linestring	Point
SDO_GTYPE	3008	3003	3002	3001
SDO_SRID	(NULL)	(NULL)	(NULL)	(NULL)
SDO_POINT	NULL	NULL	NULL	SDO_POINT_TYPE
SDO_ELEM_INFO (offset, etype, interpretation)	3 or more triplets	1 or more triplets	1 triplet	-
Header triplet	1,1007,1	-	1,2,1	-
Exterior: header triplet	1,1006,*p*	-	-	-
Exterior: element triplet	*n*,1003,1 (times the number of polygons)	1,1003,1	-	-
Interior: header triplet	*n*,2006,*p* (times the number of inner shells)	-	-	-
Interior: element triplet	*n*,2003,1 (times the number of polygons)	*n*,2003,1 (times the number of inner rings)	-	-
SDO_ORDINATE	x,y,z, (per polygon)	x,y,z, (per ring)	x,y,z (start- and endpoint)	x,y,z
n: offset *p*: number of polygons				

Table 5.3 Overview of Oracle's 3D single geometries (SDO_GEOMETRY)

Some notes should be added to this. Oracle has developed two ways of storing a polygon and a solid. When a solid is a box, the solid could be described by only 2 nodes (min-corner and the max-corner), whereas the ETYPE is different (3 instead of 1). The same applies for polygons, when a polygon is a rectangle, it could be described by only two nodes (min-corner and max-corner), whereas the ETYPE is different (3 instead of 1). In order for simplicity all faces and solids will be converted in the getGEOMETRY operations to polygons and solids, with all nodes described (ETYPE 1), even when it is a box or rectangle.
In addition to the (simple) solid, Oracle also allows the specification of a composite solid. A composite solid is a combination of *n* solids connected to each other and has been developed for easier modelling purposes only. The composite solid is used for example when difficulties arise due to the fact that the solid does not allow inner rings within a polygon, which is bounding a solid (Kothuri, Godfrind & Beinat 2007). In this research the composite solid will not be used, when an inner ring needs to be modelled in a solid, the particular polygon will be split (a restriction of one inner ring per face applies).

Four geometry operations have been developed (see for SQL-scripts, appendix I):
Operation 1) getPOINT
Composing point geometry (SDO_GEOMETRY) by the xyz values of a node
Operation 2) getLINE
Composing line geometry (SDO_GEOMETRY) by the xyz values of the startnode and endnode of an edge.
Operation 3) getPOLYGON
Combining ring ordinates to define the polygon geometry (SDO_GEOMETRY).
Operation 4) getSOLID
Combining shells of faces (using ring ordinates depending on orientation of the face) to compose a solid (SDO_GEOMETRY). For SDO_GEOMETRY no inner rings are allowed, so a face need to be split (this is done with the function dissolveINNERRINGS).[35]

In the next chapter the structure, the validation tests and the geometry operations will be tested with a data set.

[35] Every so called assistant function, like dissolveINNERRINGS, is in Appendix I, including a summary.

6. Test and evaluation

The prototype of the topological structure, the validation tests and the geometrical operations will be tested in this chapter (section 6.1). After testing an evaluation of the designed structure, validation tests and geometry operations will be put together (section 6.2).

6.1. Testing

For testing the prototype a topological correct and clean data set of the TUDelft Campus has been used, which was provided by the TUDelft.[36] The data set existed of two text files representing two tables (see figure 6.1). This model has been converted to the 3D topological model. First the data has been analyzed and inserted into the prototype, then the validation tests and geometry operations has been tested on the data set.

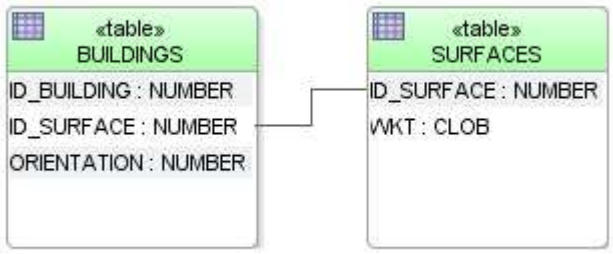


Figure 6.1 The provided data set of the campus of the TUDelft

The table BUILDINGS representing buildings as volumes, contains three columns:
- ID_BUILDING: an unique id per building (volume).
- ID_SURFACE: a reference to a face in the SURFACES table.
- ORIENTATION: the orientation (negative or positive) of the face in the volume, in such a way the normal points outward the volume.

The table SURFACES, representing the faces of the volumes, contains 2 columns:
- ID_SURFACE: a unique id per face.
- WKT: the geometry of the face (in WKT format).

The data represents the buildings of the campus of the TUDelft (Delft, Netherlands, see figure 6.2), which has about the following extent (based on the horizontal Dutch coordinate system 'Rijksdriehoeksmeting' and the Dutch vertical 'NAP' system):

	X	Y	Z
Minimum:	84936	444963	-3
Maximum:	86082	446808	90

[36] The delivered files were created with the Extrude software by Ledoux & Meijers.

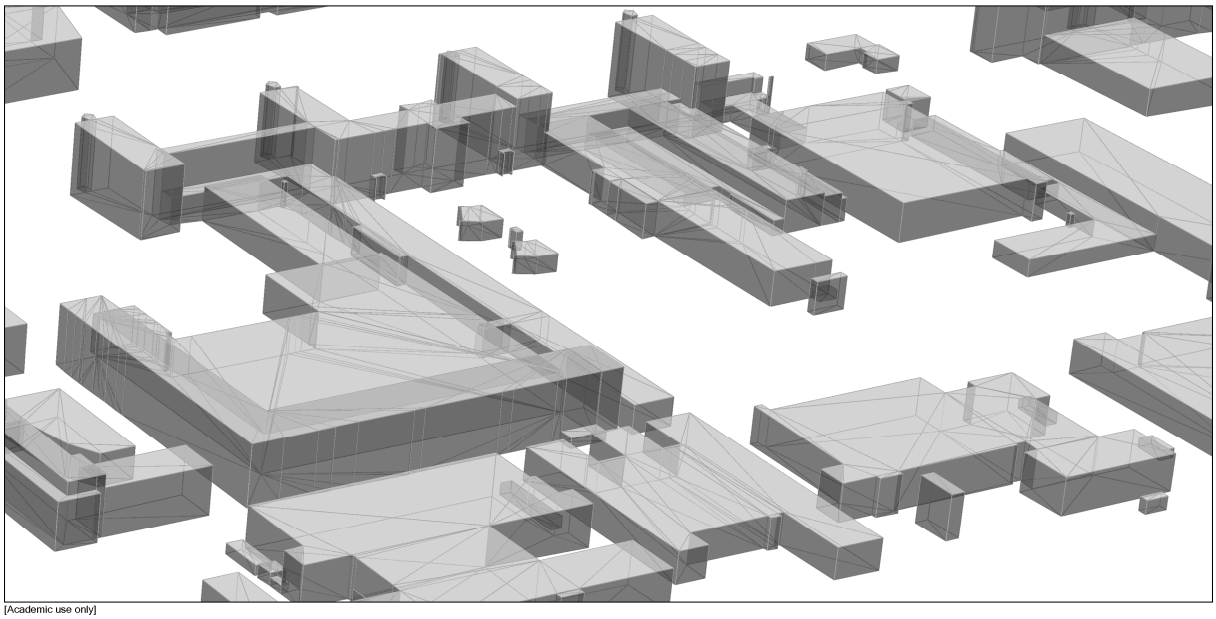


Figure 6.2 A detail of the campus data set (visualized with MicroStation)

The data has been loaded into the 3D topological structure with the help of Oracle's SQL*Loader (a bulk loader utility). The data consist of 370 (unique) volumes and 8152 (unique) faces. Furthermore 13467 (unique) edges and 5841 (unique) nodes could be derived. When analyzing the data in more detail the following information could be distracted. The volumes (buildings) are scattered around the area (campus), the air is not modelled. When clustering the volumes, 169 clusters can be distinguished. A cluster consists of buildings which are connected to each other; connected buildings share at least one face. Different clusters are separated from each other by 'air', which could be modelled. Of the 169 clusters, 130 clusters consist of only one volume, while 240 volumes are in a cluster with 2 or more volumes. These 240 volumes are clustered in 39 clusters. More specified: 15 clusters consist of 2 volumes, 9 clusters of 3 volumes, 11 clusters of 4 till 10 volumes and 4 clusters consists of 20 or more volumes. The largest cluster (cluster 381) consists of 33 buildings.
Each volume consists of an average of 22 faces, but the volumes are relative simple volumes, because several faces are in the same plane. The faces are relative simple as well. Most faces are triangles (a little more than 60%), the other faces consist of 4-7 edges. The data set is 'simple' in another way, because no inner rings and inner shells are present.
Because only 240 volumes (of the 370 volumes) are connected to another volume, usually by only one face, only very few faces (just a little more than 4%) share a volume. It also shows in the edge-face relationships that the structure is not very complex; almost 92 % of the edges is on the boundary of (just) 2 faces, and only a little bit more than 8% is on the boundary of 3 or 4 faces. In figure 6.3, buildings (distinguished by their colour) and their sharing faces (displayed in black) are visualized.

In order to visualize the data, all faces have been converted to polygons (with the operation getPOLYGON) and imported into Bentley Map (version 08.11.05.26). Bentley Map supports SDO_GEOMETRY types, including 3D types, but it does not support the solid type (GTYPE 3008). Bentley's solution is their support for the so-called 'multi-surface' and 'composite surface' (GTYPEs 3003 and 3007), as presented by Valois for Bentley Map's latest release (8.11.7) at Oracle Open World (Valois 2009). Since the solid type is not supported the faces constructing the volumes are visualized by converting them to (3D) polygons (GTYPE 3003). Polygons belonging to the same volume have the same colour, ten different colours are randomly attributed to the buildings.

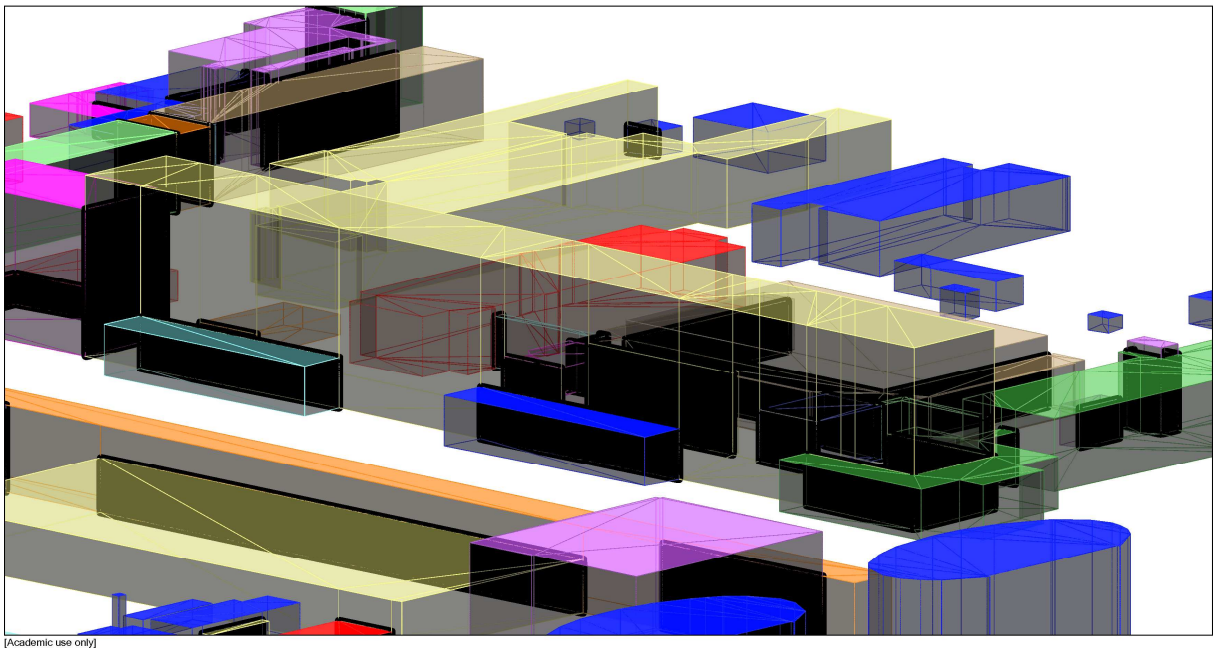


Figure 6.3 Several buildings with shared faces; the black faces are the shared ones (visualized with MicroStation)

The data has been inserted into the tables of the prototype. In order to present a full space partition, a 'mini' universe has been created for each cluster. This means the universal volume has 169 inner shells.
The validation tests and geometry operations are tested on the whole campus data set and on the largest cluster (cluster 381) with 33 buildings (see figure 6.4). This clusters consists of 33 volumes and the universal volume with one inner shell, 893 faces, 1455 edges and 594 nodes.

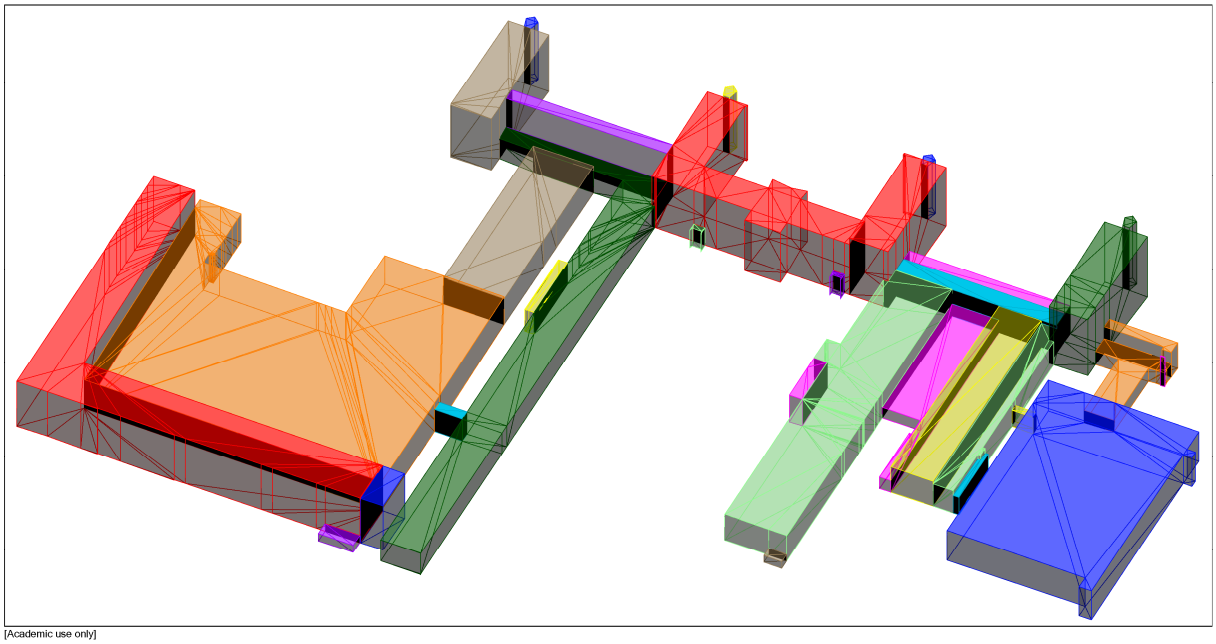


Figure 6.4 Cluster381 of the Campus data set (visualized with MicroStation)

Although performance (in time) is out of scope of this research, still the duration of the tests have been analyzed. In order to get an idea about the performance, but keep in mind the PL/SQL-scripts are not very fast and efficient. The results of the tests are presented in table 6.1.

<table>
<tr><th></th><th colspan="2">Duration (seconds)</th><th>Result</th></tr>
<tr><td></td><td>Campus</td><td>Cluster381</td><td></td></tr>
<tr><td colspan="4">Validation tests</td></tr>
<tr><td>1</td><td>1,0</td><td>0,5</td><td>Ok</td></tr>
<tr><td>2</td><td>0,5</td><td>1,0</td><td>Ok</td></tr>
<tr><td>3</td><td>0,5</td><td>0,5</td><td>Ok</td></tr>
<tr><td>4a</td><td>70,5</td><td>2,0</td><td>Ok</td></tr>
<tr><td>4b</td><td>17,0</td><td>2,0</td><td>Ok</td></tr>
<tr><td>5a</td><td>n.a.</td><td>n.a.</td><td>No inner rings in structure</td></tr>
<tr><td>5b</td><td>30,5</td><td>0,5</td><td></td></tr>
<tr><td>6</td><td>56,0</td><td>3,0</td><td>For different tolerance values, different results:
<table>
<tr><th></th><th colspan="2">Campus</th><th colspan="2">Cluster381</th></tr>
<tr><td>Tolerance (m)</td><td>Not flat</td><td>Flat</td><td>Not flat</td><td>Flat</td></tr>
<tr><td>0,005</td><td>7993</td><td>159</td><td>775</td><td>118</td></tr>
<tr><td>0,01</td><td>7961</td><td>191</td><td>773</td><td>120</td></tr>
<tr><td>0,5</td><td>6280</td><td>1872</td><td>588</td><td>305</td></tr>
<tr><td>1,0</td><td>4845</td><td>3307</td><td>470</td><td>423</td></tr>
<tr><td>5,0</td><td>1367</td><td>6785</td><td>175</td><td>718</td></tr>
<tr><td>10,0</td><td>375</td><td>7777</td><td>58</td><td>835</td></tr>
</table></td></tr>
<tr><td>7a</td><td>6362,0</td><td>335,0</td><td>face 2362 and 5197 have an intersection with another face of the same volume (0). A closer look concludes edge 5619 (of inner shell 463 of volume 0) interacts with face 2362 and edge 7533 (of inner shell 399 of volume 0) interacts with face 5197.</td></tr>
<tr><td>7b</td><td>n.a.</td><td>n.a.</td><td>No inner rings in structure</td></tr>
<tr><td>7c</td><td>n.a.</td><td>n.a.</td><td>No inner shells (except for the universal volume, but those shells are not checked in this test).</td></tr>
<tr><td colspan="4">Geometry operations</td></tr>
<tr><td>getPOINT</td><td>16,5</td><td></td><td>For the first 5000 nodes</td></tr>
<tr><td>getLINE</td><td>19,5</td><td></td><td>For the first 5000 nodes</td></tr>
<tr><td>getPOLYGON</td><td>41,5</td><td></td><td>For the first 5000 nodes</td></tr>
<tr><td>getSOLID</td><td>51,8</td><td></td><td>For all 370 solids</td></tr>
</table>

Table 6.1 Test results for total campus data set

Three tests (5a, 7b and 7c) could not be tested because no inner shells (except for 169 ones in the universal volume) and no inner rings are present in the structure. These tests are tested on the handmade mini data set, which consists of only 10 volumes (plus one universal volume), 58 faces, 110 edges and 67 nodes (see figure 6.5).

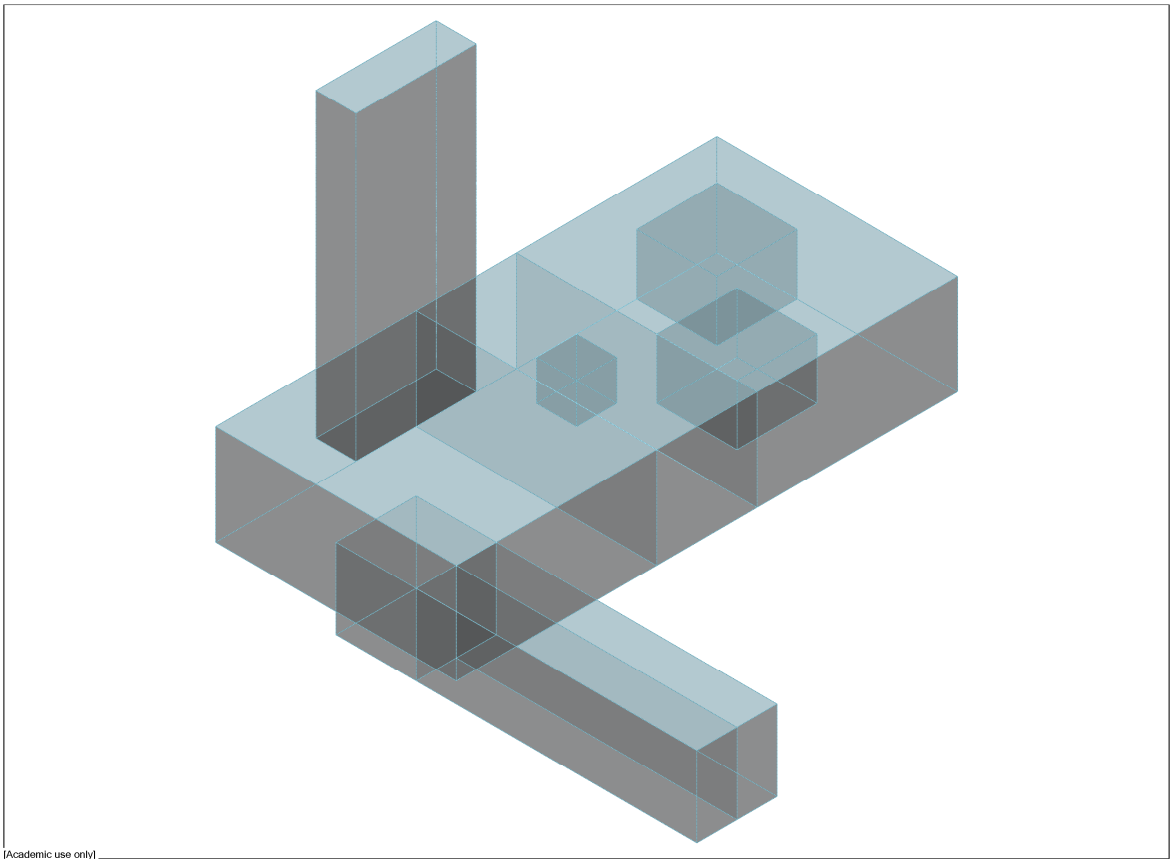


Figure 6.5 Handmade mini data set (visualized with MicroStation)

To check the credibility of the validation tests. All primitives are tested with Oracle's SDO_GEOM.VALIDATE_GEOMETRY_WITH_CONTEXT after being converted to a geometry. Oracle took 4061 seconds (including the function getSOLID) to validate the 370 solids, 137 seconds for all polygons, 42 seconds for all lines and 16 seconds for all points. Two errors has been raised both solid 7 (175 faces) and 3 (109 faces) received error '54502: solid not closed'. This means the solid geometry was not closed i.e., faces of solid are not 2-manifold due to incorrectly defined, oriented, or traversed line segment because each edge of a solid must be traversed exactly twice, once in one direction and once in the reverse direction (www.error-code.org.uk). When analyzing the error message, it seems it does not indicate which edge is not 'traversed' twice, while the message should indicate this. The raised error message: '54502 Point:0,Edge:0,Ring:0,Polygon:0,Comp-Surf:1'. The reason for this error is unclear.

Based on the above observations the following conclusions could be drawn. It seems most tests are within acceptable time limits, only 7a 'intersections' is unacceptable. It is a complex test, in which many calculations must be done. Before the geometrical function of Oracle tests for any interaction, first a geometrical primitive has to be constructed. This is not a very efficient way. For test 7a it seems the best way at this moment, because intersections are purely geometrical. Since there is no outcome for the tests involving inner rings or inner shells (tests 5a, 7b and 7c), these conclusions will be based on the mini data set. Test 5a 'proper orientation of inner ring' does not involve complex geometrical calculations and will therefore probably perform well. On the other hand tests 7b and 7c 'every inner boundary must be inside its accompanying outer boundary' makes also use of Oracle's SDO_ANYINTERACT, like test 7a, and are both estimated to perform a little worse. Although the performance time will not be close to the more than 6000 seconds test 7a took, because much less interactions need to be checked in tests 7b and 7c.

Furthermore it is detected that three tests are not fully developed (yet):

- Test 1c: The rare situation when 3 or more faces have the same main characteristics (number of rings, number of edges, and the edges with the lowest and highest id) will be detected but not tested on equal edge collections.
- Test 6a: A flat face is calculated by its centroid, instead of calculating all nodes.
- Test 7a: The check for node-in-face and edge-in-face is not added yet (based on the explicitle stored relationship).
- Test 7c: the check for inner shells of the universal volume not intersecting each other and not being inside each other, must be added.

Besides evaluating the results and performance of the tests and operations, next an evaluation on data storage will follow. A comparison with the TEN-structure is made. Penninga (2008, chapter 7) has analyzed the characteristics, of the same campus data set. Because of the lack of information about the tetrahedra of the test data set, the numbers of Penninga will be used for comparing the prototype with the TEN structure. The numbers of the data sets are summarized in table 6.2. Keep in mind that this table presents a lot of uncertainties; many numbers are based on estimations.

	Buildings	**Tetrahedrons/ Volumes**	**Triangles/ Faces**	**Constrained triangles**	**Edges**	**Nodes**
TEN (based on Penninga)	306	131457	350230*[1]	87316	27737*[2]	23260
3D topological structure (based on Penninga)	306	307	4503	n.a.	10878*[3]	8182
3D topological structure (based on test data set)	370	371	8152	n.a.	13467	5841
*1 the number of triangles is estimated based on the number of tetrahedra and constrained triangles. *2 the number of edges is estimated based on the average number of tetrahedra a node is part of. *3 The number of edges is estimated based on the average number of nodes per building.						

Table 6.2 Analysis of the campus data set

Based on these estimated numbers, a calculation on storage is made. This calculation is very arbitrary as based on several estimations and arbitrary parameters, like number of tables, rows and columns and the used data types. The estimation of the numbers (table 6.2) are made for this particular data set. This means for data sets with the same level of detail (the source data of this data set is the Dutch large scale base map, 1:1000). The relationship between the numbers is different for data sets within another level of detail. For a data set with more detail the proportion between the numbers is different than the proportion of the numbers for data sets with less details, as is shown by Penninga (2008, chapter 7). For both structures (3D topological structure and TEN) a full space partition has a price on the number of primitives. As stated by Penninga, a little more than 70% of the tetrahedra is used for modelling the air. Concluded from the test data set (only) a little more than 31% of the shells is used for the universal volume.

When storing the TEN the classic way, based on a relational implementation of Pilouk (Zlatanova, Abdul Rahman & Shi 2004), three topological tables will be populated:
- TRIANGLES: 350230 rows; 6 columns (only number data types).
- EDGES: 27737 rows; 3 columns (only number data types).
- NODES: 23260 rows; 4 columns (only number data types).

Total of 401.227 rows
Total of 2.277.631 populated fields

When storing the TEN the simplicial complex approach (Penninga 2008), only one table is populated (and 5 views are derived):
- Tetrahedron: 131457 rows; one column (nvarchar2(300) data type).

Total of 131.457 rows
Total of 131.457 populated fields

When populating the 3D topological structure (based on the estimation of Penninga), seven tables (and 2 views) are used:
- NODE: 8182 rows; 4 columns (numbers)
- EDGE: 10878 rows; 3 columns (numbers)
- RING2EDGE: 38073 rows; 3 columns (numbers and varchar(1))[37]
- RING: 4503 rows; 3 columns (numbers and varchar(1))
- FACE2SHELL: 4503 rows; 3 columns (numbers)
- SHELL2VOLUME: 446 rows; 3 columns (numbers and varchar(1))[38]
- VOLUME: 307 rows; 1 column (number)

Total of 66.892 rows.
Total of 208.224 populated fields

When populating the 3D topological structure (based on the used data set), seven tables (and 2 views) are used:
- NODE: 5841 rows; 4 columns (number)
- EDGE: 13467 rows; 3 columns (numbers)
- RING2EDGE: 28071 rows; 3 columns (numbers and varchar(1))
- RING: 8152 rows; 3 columns (numbers and varchar(1))
- FACE2SHELL: 8152 rows; 3 columns (numbers)
- SHELL2VOLUME: 539 rows; 3 columns (numbers and varchar(1))
- VOLUME: 371 rows; 1 column (number)

Total of 64.593 rows.
Total of 198.878 populated fields

Although it is necessary to be careful with drawing conclusions from the above summaries, some observations draw attention. For a more balanced comparison the 3D topological structure based on the estimation of Penninga will be used, because it is based on the same source. First there is a large difference in the number of rows and the number of tables. The simplicial complex based TEN has only one table, while the 3D topological structure has 7 tables. The classic TEN has 401.227 rows while the 3D topological structure has (only) 66.892 rows. When combining those two parameters by counting the number of populated fields (rows per table times the number of columns), the simplicial complex based TEN has 131.457 populated fields, while the classic TEN has 2.277.631 populated fields. The 3D topological structure is closer to the simplicial complex based TEN with 208.224 populated fields.

These observations are based on storage numbers, earlier observations were based on the ability of conversion to geometries and validation possibilities. These are only a few aspects of many others which will contribute to a final conclusion about a structure. Aspects like the usability of neighbourhood operations and the possibilities for updating a structure. For these aspects only some thoughts could be defined. Since relationships between primitives (1 dimension lower or higher) are stored explicitly it is expected to be relatively easy to design neighbour functions on the structure. The same reasons apply for edits, since all primitives and their relationships are explicitly stored, a primitive could be easily redefined, although more relationships stored means more updates needed.

Based on the above conclusion the advantages and disadvantages of the designed topological approach will be discussed in the next section.

6.2. Evaluation

The advantages and disadvantages of the 3D topological structure will be pointed out and possible extensions and/or adjustments are suggested.

[37] Estimation; based on: every edge is on the boundary of an average of 3,5 faces.
[38] Estimation; based on the test data set.

This evaluation is based on the theoretical tests (section 5.3), the tests performed with the campus data set complemented with the tests performed with the handmade mini data set.

First, two remarks about the evaluation method. Efficiency (in time) is not the goal of this research, therefore the outcome on the duration of the tests is only a rough evaluation. The second remark, the test data set (of the TUDelft campus) did not include all characteristics which should be tested. The main lacks were in inner rings and inner shells as well as in complex primitives, especially faces. A complex face, for example is a face which is not convex and consists of many edges.

The principle of designing the validation tests was to use existing Oracle functions when they are at hand. These functions are developed for single geometries, therefore they are suitable for geometry related tests. Although it means the involved primitives must be converted to geometrical primitives first. Within the validation tests two of these Oracle functions are used. SDO_ANYINTERACT detects any undefined geometrical intersection between primitives and is used in tests 7a, 7b and 7c.

It can be concluded that the geometrical related tests, grouped in the 'geometry block', are the most difficult and inefficient tests. Which is not very surprisingly. It is clear that these tests are related to a lot of computing. Another reason is the third dimension. Although, this applies to all tests, it is especially relevant for the orientation and geometry related tests. Checking the orientation of faces in one plane (2D) is much easier than testing the orientation of faces in 3D and testing the orientation of volumes. The same applies for intersections which is much more complicated in 3D and for testing for a 'contiguous area/volume', which is easy in 2D, but very complicated in 3D.

Furthermore the modelling approach also influences the efficiency of validation. A TEN is easier to validate than the used 3D topological structure, because of the simple primitives (triangles and tetrahedra) in a TEN. For example a TEN does not permit any singularities and an edge is always used exactly two times within one volume (tetrahedron), therefore testing for a contiguous volume is quite easy. Also testing for intersections is easier and a TEN does not have any inner- shells or rings. A disadvantage of a TEN is the number of volumes (tetrahedra), which is large. Other disadvantages of a TEN are the need for storing constrained primitives and the large part which is taken by 'space' (e.g. air). On the other hand the large storage space a TEN takes has been reduced by the approach of Penninga.

Remarkable within this 3D topological structure, compared to other 3D topological structures, is the explicit storage of rings and shell within the chain of boundaries. The advantages of storing these boundaries explicitly, is related to the crucial role the boundaries play within a topological structure. Shells and rings are used in many validation tests and the geometry operators. Among others in test 3 'one outer boundary', test 4 'closed boundaries' and test 5 'proper orientation' but also in the operations getPOLYGON and getSOLID. The disadvantages of storing the rings and shells explicitly are more tables and more references

Based on the above discussion the advantage, disadvantages and recommendations will be listed.

The advantages

- The structure is transparent and compact. The structure is clearly set out and it is easy to go along all the relationships. The main characteristics (like boundaries and orientation) are modelled. This makes it easy to extract information (like geometries or neighbours) out of it and to maintain the structure.

- Most validation tests could be performed easily; most tests could be done based on topological references (instead of using geometry), complex geometrically based validation tests are reduced to a minimum. This means no tests are needed on intersections between volumes (only self-intersections of volumes) and no tests are needed on self-intersections of faces. For checking the orientation of shells, it is only needed to tested one face per inner shell.
- Fast conversion to geometrical primitives.
- Structure is easy to populate.
- The structure is easy to maintain.
- The number of possible situations for failing the 'difficult' test 8 'contiguous volume' has been reduced.

The disadvantages
- It is a full space partition therefore everything need to be modelled (but at the same time this full partition is also the basis of the strength of the structure).
- Some derivations could be long (but therefore the views are added).
- Inner rings are allowed, which makes it hard to deal with inner rings when realizing a geometrical solid (in SDO_GEOMETRY). At the moment only one inner ring per face is allowed.
- Some geometrical related tests are complicated, like testing contiguous volumes and intersections (but they are reduced to a minimum).

Recommendations: possible extensions and/or adjustments
- Extract the rings and shells in such a way that edges are directly connected to faces and faces directly to volumes.
- Updating functionality
- Add tolerance values.
- Add features to the model.
- Remove the restriction of only one inner ring per face.
- Efficient algorithms (make use of spatial indices)
- Implement test 8 'contiguous volume'
- Convert algorithms to dynamic SQL
- Adjust tests 1c, 6a, 7a and 7c, they are not fully developed yet

Conclusion

A prototype of a 3D topological structure with validation and conversion functionality has been implemented within Oracle Spatial. It is a linear structure of a volume partition, which is ISO 19107 compliant.
This main goal of the research has been reached by answering several sub questions, which were defined in section 1.3. First the requirements for a 3D topological structure were set. These requirements were defined based on literature and by analyzing several existing 2D and 3D topological structures. 2D topological structures are well developed, while 3D topological structures are underdeveloped. Although several 3D topological structures exist, most mainstream geo-DBMSs do not support 3D topological structures, only 1Spatial has recently developed a 3D topological structure.

Whereas 2D topological structures seem clearly set out, defining 3D topological structures is more complex. Therefore a simple 'extension' from 2D topological structures to 3D topological structures is not possible. For example the widely used winged edge structure in 2D, can not be simply applied in 3D. An edge in 3D can have many 'wings', while in 2D, each edge always has four 'wings' (a next left, next right, previous left and previous right). Also orientation of faces in 3D, can not be defined as (counter)clockwise, like in 2D. Although the characteristics of a topological structure are based on the same aspects in 2D and in 3D, the requirements are much more complex in 3D. These aspects are: dimension, partition (including the universe), primitives (including their relationships), orientation, singularities and geometrical realization. For the 3D topological structure the requirements are set and based on four primitives related to each other by their (co)boundary relationships within a full space partition.

Other examples of 3D topological structures (Radius topology, 3D FDS, SSS and TEN), which are discussed in this research, show quite some differences. Some structures maintain a 3D primitive while others do not. Orientation, which is more complicated in 3D than in 2D, is stored in different ways, the same applies for geometry and singularities. Some structures have quite some redundancy which does not agree with the main advantages of a topological structure in general. Other structures have only a few relationships stored explicitly, which makes maintaining the structure hard. Therefore the right balance between usability and efficiency is needed, where the data is easy accessible and useable (no complicated derivates) and the structure is still compact, without redundancy.

The structure makes efficient use of the advantages of a topological structure in general, like less redundant storage, maintain data consistency and efficient queries for relationships. Single geometries have their own advantages therefore it is very important to be able to use both structures and convert topological primitives into geometrical primitives. No topological structure is the single topological structure suitable for all applications. The strong argument for this 3D topological structure is its robustness and accessibility. It has all its main characteristics explicitly stored in an efficient way. Therefore validation, analysis and realizing geometries are efficient. On the other hand the structure remains compact and is not very redundant, although it has quite a lot of tables (7). The structure is easy to populate and easy to maintain.

In order to perform any analysis and maintain the structure it is very important to know that the structure is valid. Therefore a validation function has been developed and implemented on the topological structure. First, existing functions have been analyzed. There is no single definition for a 3D primitive, not for single geometries and not for topological structures. Based on several existing definitions for 3D single geometries and the requirements for the structure, a set of validation rules have been defined for the valid 3D topological structure. A valid topological structures means a valid primitive (volume) and valid relationships between the volumes. Several 2D topological validation functions exist and a few 3D single geometry validation functions exist, but no 3D

topological validation function. On the topological structure validation functionality has been implemented.

Recommendations for further research will be on elaborating the structure with features. and edit functionality. Improve current structure and validation function by more efficient algorithms, the implementation of a test for a ‘contiguous volume’.

References

Abdul Rahman, A., S. Zlatanova & M. Pilouk (2001), *The 3D GIS software development: global efforts from researchers and vendors*.

Arens, C., J. Stoter & P. van Oosterom (2005), *Modelling 3D spatial objects in a geo-DBMS using a 3D primitive*

Baumgart, B.G. (1975), *A polyhedron representation for computer vision*. National Computer Conference, pp 589-596.

CEN/TC287 (2005), *International standard EN ISO 19107, Geographic information - spatial schema.*

De la Losa, A. & B. Cervelle (1999), *3D topological modelling and visualisation for 3D GIS. In: computers and graphics*.

Ellul, C. (2007), *Functionality and performance - two important considerations when implementing topology in 3D.*

Ellul, C., M. Haklay & T. Bevan (2005), *Deriving a generic topological data structure for 3D data.*

ESRI (2009), *ARcGIS Desktop Help 9.3.*

Groger, G. & L. Plumer (2005), *How to get 3D for the price of 2D - topology and consistency of 3D urban GIS.*

ISO/TC211 (2003), *International standard ISO 19107, Geographic information - spatial schema.* ISO: Geneva, Switzerland.

ISO/TC211 (2004), *OpenGIS Implementation specification for geographic information - simple feature access – part 1: Common architecture / part 2: SQL option*.

ISO/TC211 (2010), *ISO/DIS 19152 Geographic information - Land Administration Domain Model (LADM) - Enquiry.* ISO: Geneva, Switzerland.

Kazar, B.M., R. Kothuri, P. van Oosterom & S. Ravada (2008), *On valid and invalid three-dimensional geometries.*

Khuan, C.T., A. Abdul Rahman & S. Zlatanova (2007), *New 3D data type and topological functions for geo-DBMS.*

Khuan, C.T., A. Abdul Rahman & S. Zlatanova (2008), *3D solids and their management in DBMS.*

Kothuri, R., A. Godfrind & E. Beinat (2007), *Pro Oracle Spatial for Oracle Database 11g.*

Molenaar, M. (1990), *A formal data structure for 3D vector maps*. In: Proceedings of EGIS'90, Vol. 2. Amsterdam, The Netherlands, pp. 770–781.

Murray, C. et.al. (2008a), *Oracle spatial developer's guide 11g release 1 (11.1).*

Murray, C. et.al. (2008b), *Oracle spatial topology and network data models developer's guide 11g release 1 (11.1).* Oracle USA, Inc.: Redwood City, CA, USA.

O'Connor, J.J. & E.F. Robertson (1996), *History topic: A history of Topology*.

Oosterom, P. van, W. Quak & T. Tijssen (2004), *About invalid, valid and clean polygons.*

Oosterom, P. van, J. Stoter, W. Quak & S. Zlatanova (2002), *The balance between geometry and topology.*

Oosterom, P. van, W. Vertegaal & M. van Hekken (1994), *Integrated 3D modelling within a GIS.* AGDM'94: Delft, Netherlands.

Penninga, F. (2008), *A simplicial complex-based solution in a spatial DBMS.*

Pilouk, M. (1996), *Integrated Modelling for 3D GIS*. Ph.D. thesis, ITC Enschede, Netherlands.

Quak, W., J. Stoter & T. Tijssen (2003), Topology in spatial DBMSs. Delft: Delft University of Technology.

Reeve, D. & J. Petch (1999), *GIS organisations and people, a socio-technical approach.*

Ryden, K. (2005), *OpenGIS Implementation specification for geographic information - simple feature access – part 1: Common architecture / part 2: SQL option*. OGC: Wayland (MA), USA.

Thompson, R. & P. van Oosterom (2009), *Mathematically provable correct implementation of integrated 2D and 3D representations.*

Valois, F. (2009), *Bentley Map and Oracle Spatial 3D helps advancing GIS for infrastructure*. Presented at Oracle Open World (October 2009, San Francisco USA).

Watson, P.J., M.J. Martin & T.D.c. Bevan (2008), *WO 2008/138002 A1 Three-dimensional topology building method and system.* World Intellectual Property Organization: Geneva, Switzerland.
Zlatanova, S. (2000), *3D GIS for urban development 2000.*
Zlatanova, S. (2008), *Freeform data types in spatial database.*
Zlatanova, S., A. Abdul Rahman & M. Pilouk (2002), *Trends in 3D GIS development.*
Zlatanova, S., A. Abdul Rahman & W. Shi (2004), *Topological models and frameworks for 3D spatial objects.*

Websites

1Spatial: http://www.1spatial.com/
PostGIS Tracker and Wiki - PostGIS Geospatial Objects for PostgreSQL: http://trac.osgeo.org/postgis/
Error-code.org.uk: http://www.error-code.org.uk/

Appendix I – PL/SQL-scripts

Contents:

- Summary of the scripts
- Scripts
  - Create structure
  - Validation tests (1-7)
  - Geometry operations
  - Assistant functions

Summary of the scripts

Test 1) unique primitives

Test 1a) unique nodes
Tables involved: NODE table
Functions involved: none
Primitives involved: nodes
Feasibility: simple
Remark: none
Summary: all nodes are tested for unique x,y,z values.

Test 1b) unique edges
Tables involved: EDGE table
Functions involved: none
Primitives involved: nodes and edges
Feasibility: simple
Remark: none
Summary: all edges are tested for a unique startnode-endnode combination (no edges with the same startnode and endnode), a startnode not being the same as the endnode and a startnode-endnode combination not equalling an endnode-startnode combination.

Test 1c) unique faces
Tables involved: RING2EDGE and RING tables
Functions involved: none
Primitives involved: edges and faces (including rings)
Feasibility: simple
Remark: when 3 or more faces have the same main characteristics they are not tested on equal edge collections (although they are detected as 'not tested').
Summary: All faces are tested for uniqueness, first a global check is done on 4 main characteristics of a face (the number of rings, the number of edges, the min. edge (edge with the lowest id) and the max. edge (edge with the highest id)). When faces with the same characteristics are detected, they are checked on uniqueness for their whole edge collection. When they have the same edge collections they are tested 'not unique'.

Test 2) primitive references

Test 2a) node-edge references
Tables involved: NODE and EDGE tables
Functions involved: none
Primitives involved: nodes and edges
Feasibility: simple
Remark: none
Summary: Each node is tested for isolation; each node in the node table must exist in the EDGE table. By the foreign keys on the start- and endnode in the EDGE table ('startnode_fk' and 'endnode_fk'), nonexistent nodes are not possible.

Test 2b) edge-face references
Tables involved: EDGE, RING2EDGE and RING tables
Functions involved: none
Primitives involved: edges and faces (including rings)
Feasibility: simple
Remark: none
Summary: Each edge is tested for isolation: each edge in the EDGE table must exist in the RING2EDGE table and each ring is tested for nonexistence. Each ring in the RING table must exist in the RING2EDGE table. By the foreign keys on edges and rings in the RING2EDGE table ('edge_fk' and 'ring_fk'), nonexistent edges and isolated rings are not possible. Furthermore each ring will be tested for 3 or more (unique) edges and each

face (with an inner ring) will be tested for rings not sharing a same edge'. Finally each face will be tested for more than 1 node which is used more than once in a face.
Background: A ring could not refer more than one time to the same edge (by the primary key RING2EDGE_pk), but inner and outer rings (of one face) could refer to the same edge (which is not allowed). This test tests (for reasons of simplicity) all edge references in a face (including edges of the same ring, while this could not occur by the RING2EDGE_pk).

Test 2c) face-volume references
Tables involved: RING, FACE2SHELL and SHELL2VOLUME tables
Functions involved: none
Primitives involved: faces and volumes (including shells)
Feasibility: simple
Remark: none
Summary: Each face and shell is tested for nonexistence; each face in the FACE2SHELL table must exist in the RING table and each shell in the SHELL2VOLUME table must exist in the FACE2SHELL table. By the foreign keys on the shell and face references ('face_fk', 'shellpos_fk' and 'shellneg_fk') isolated faces and shells are not possible. Furthermore each shell in a volume is tested on existing of four or more (unique) faces. Finally each volume is tested for unique faces (a face may only occur once in a volume (including inner and outer shells). A face can not refer twice to the same shell (one neg. and one pos. reference) and also it is not allowed for faces to refer to the inner and outer shell of one volume.

Test 3) each face/volume exists of one outer boundary

Test 3a) each face exists of one outer ring
Tables involved: RING table
Functions involved: none
Primitives involved: faces (including rings)
Feasibility: simple
Remark: none
Summary: Each face is tested for exactly 1 outer ring (no more and no less than 1 outer ring). allowed. For the moment a test is added for a maximum of 1 inner ring.

Test 3b) each volume exists of one outer shell
Tables involved: SHELL2VOLUME table
Functions involved: none
Primitives involved: faces and volumes (including shells)
Feasibility: simple
Remark: none
Summary: Each volume is tested for exactly 1 outer shell (no more and no less than 1 outer ring). The universal volume is an exception and exists of only (an) inner shell(s). the universal volume is tested for its presence.

Test 4) closed boundaries

Test 4a) closed ring
Tables involved: RINGORDINATES view
Functions involved: getRINGordinates
Primitives involved: nodes, edges and faces (including rings)
Feasibility: simple
Remark: none
Summary: The test for a closed ring is taken care of by the function getRINGordinates, which is displayed in the view RINGORDINATES. When the function returns a NULL value

the test is not passed (see getRINGordinates). The function tests if the ring is closed (end node equals start node) and tests the orientation of each edge (in the ring).

Test 4b) closed shell
Tables involved: RING2EDGE, RING, FACE2SHELL and SHELL2VOLUME tables and NODE2SHELL view
Functions involved: none
Primitives involved: edges, faces and volumes (including rings and shells)
Feasibility: medium
Remark: none
Summary: Each edge, involved in the shell, is tested for an equal distribution of negative and positive references.
Background: When checking the distribution of negative and positive references in each shell, consideration is needed for the orientation of faces in shells and the orientation of edges in a face (positive face, with positive edge and negative face with negative edge is a positive reference and a negative face with a positive edge and a positive face with a negative edge is a negative reference).

Test 5) Proper orientation

Test 5a) each inner ring must have a proper orientation
Tables involved: RING table
Functions involved: getNORMAL
Primitives involved: edges and faces (including rings)
Feasibility: medium
Remark: a tolerance value (0.01) is added at the comparison of the normal values
Summary: The normal of every inner ring is compared with the normal of the accompanying outer ring. The normal is calculated from the ordinate-array of the ring (see function getNORMAL). A face is tested 'not proper oriented' when the normal of an inner ring is not opposite to the normal of the accompanying outer ring. A face will also not pass this test when the inner ring has a different slope than the outer ring (test 6a).

Test 5b) each inner shell must have a proper (inward) orientation
Tables involved: NODE, RING, SHELL2VOLUME table NODE2SHELL view
Functions involved: getNORMAL
Primitives involved: nodes, edges and faces (including rings and shells)
Feasibility: simple
Remark: none
Summary: For each inner shell a face with the lowest z-value is selected, which is not vertical (along the y-axis). When the z-value of the normal is negative the face, and thus the inner shell, is oriented properly.

Test 6) planar faces

Test 6a) planar faces
Tables involved: RING and FACE2SHELL tables
Functions involved: none
Primitives involved: nodes and faces (including rings)
Feasibility: medium
Remark: When a face is not flat but still has its centroid on the plane, will not be detected as INVALID.
Summary: Each ring of a face will be tested on planarity by calculating its centroïd (the mean of al x, y and z values) and calculating the distance from the centroïd to the plane. For inner rings an extra test is added by calculating the distance of the centroid to the plane of the outer ring (test for 'sunken' inner rings). A tolerance value (within the distance) is admitted, which could be adjusted depending on the data set.

Test 7) no intersections

Test 7a) no self intersecting volumes
Tables involved: RING2EDGE, RING, FACE2SHELL and SHELL2VOLUME tables, the NODE2SHELL view and temporarily the INTERACTION table
Functions involved: getLINE, getPOLYGON and SDO_ANYINTERACT
Primitives involved: edges, faces and volumes
Feasibility: medium
Remark: none
Summary: All volumes are checked on edge-face intersections. All edges of a volume are collected and temporarily stored in the INTERACTION table, for each edge the geometry is added. Next oracle's function SDO_ANYINTERACT will check per face (of that volume) if any edge (the edges being on the boundary of both faces are left out) interacts with that face, except for a node sharing touch. Singularities node-in-face or edge-in-face are valid interactions (check with the columns node.node_in_face_id_ref and edge.edge_in_face_id_ref)[39].

Test 7b) every inner ring must be inside the accompanying outer ring
Tables involved: RING table and temporarily the INTERACTION table
Functions involved: SDO_ANYINTERACT
Primitives involved: nodes and faces (including rings)
Feasibility: simple
Remark: none
Summary: The geometry of all edges of the inner ring are checked for interaction with the geometry of the accompanying outer ring with the help of oracle's SDO_ANYINTERACT function. When all edges have 'an' interaction, the inner ring is tested 'inside' the outer ring.

Test 7c) every inner shell must be inside the accompanying outer shell
Tables involved: RING, FACE2SHELL and SHELL2VOLUME tables and temporarily the INTERACTION table
Functions involved: SDO_ANYINTERACT and getSHELL
Primitives involved: nodes, faces and volumes (including rings and shells)
Feasibility: simple
Remark: The test for intersecting inner shells of the universal volume, is not added yet.
Summary: The geometry of all faces of the inner shell is checked for (any) interaction with the geometry of the accompanying outer shell with the help of oracle's SDO_ANYINTERACT function. Except for the inner shell of the universal volume. When the universal volume has more than 1 inner shell, the inner shells must be tested for intersecting each other and being inside each other.

[39] Must still be added.

Geometry operations

Operation) getPOINT
Tables involved: NODE table
Functions involved: none
Input/output: node_id / SDO_GEOMETRY
Primitives involved: nodes
Feasibility: simple
Remark: none
Summary: a point geometry is extracted from the x,y,z values.

Operation) getLINE
Tables involved: NODE and EDGE tables
Functions involved: none
Input/output: edge_id / SDO_GEOMETRY
Primitives involved: nodes and edges
Feasibility: simple
Remark: none
Summary: a line geometry is extracted from the x,y,z values of the start- and endnode.

Operation) getPOLYGON
Tables involved: RING table
Functions involved: none
Input/output: face_id / SDO_GEOMETRY
Primitives involved: edges and faces (including rings)
Feasibility: simple
Remark: none
Summary: The faces are checked for rings, since the orientation of rings/faces are stored randomly it is not necessary to take any consideration about orientation. This functions collects all rings of a ring and merges them together in the ordinate-array and constructs the array-info.

Operation) getSOLID
Tables involved: RING, FACE2SHELL and SHELL2VOLUME tables
Functions involved: dissolveINNERRING
Input/output: volume_id / SDO_GEOMETRY
Primitives involved: faces and volumes (including rings and shells)
Feasibility: medium
Remark: none
Summary: The ordinate-array and array-info are constructed. First the outer shell of the volume is collected and its faces, considering the orientation of each face. Then (if present) the inner shells are collected and all the faces involved, considering their orientation. Finally when an inner ring is present the face will be splitted into two faces by the help of the function dissolveINNERRING.

Assistant functions

Function) getRINGordinates
Tables involved: NODE, EDGE and RING2EDGE tables
Used in tests/operations: test 4a
Functions involved: none
Input/output: ring_id / sdo_ordinate_array
Primitives involved: nodes and edges (including rings)
Feasibility: medium
Remark: none
Summary: This function is used to populate the view RINGORDINATES and includes a check on the orientation of the edges in a ring.

Function) getNORMAL
Tables involved: RING table
Used in tests/operations: test 5a and 5b
Functions involved: none
Input/output: ring_id / normal_type (is varray (3) of number)
Primitives involved: faces (including rings)
Feasibility: simple
Remark: the output is an user defined data type
Summary: The function calculates the normal of the input ring (ring_id), the normal is calculated by the cross product of two vectors of the ring. The two vectors are calculated from the ordinate-array of the ring. The first two edges are derived, when these two edges are parallel two other edges are derived (until two edges are found which are not parallel). Finally the normal is normalized in order to delete the influence of the length of an edge.

Function) getSHELL
Tables involved: RING and FACE2SHELL tables
Used in tests/operations: test 7c
Functions involved: dissolveINNERRING
Input/output: shell_id / SDO_GEOMETRY
Primitives involved: edges and faces (including rings and shells)
Feasibility: simple
Remark: none
Summary: This function is derived from the operation getSOLID and will construct the geometry of a shell (irrespective of being an inner or outer shell) as a solid with one outer shell.

Function) dissolveINNERRING
Tables involved: EDGE, RING2EDGE and RING tables
Used in tests/operations: getSHELL and getSOLID
Functions involved: none
Input/output: face_id / ring_ordinate1 and ring_ordinate2
Primitives involved: edges and faces (including rings)
Feasibility: complex
Remark: This function works only for faces with one inner ring (faces with more than 1 inner rings are detected but not constructed as faces without inner rings).
Summary: Inner rings need to vanish by splitting the polygon in to two (or when needed more) polygons in such a way no inner rings are needed. In this procedure for the outer ring and inner ring both the max and min nodes are selected based on the x-ordinate (when all x-ordinates are equal, the y-ordinate is used and possibly the z-ordinate). Then the new face is constructed by following this route: from the max node on the outer ring following the outer ring (in the proper direction) till the min node on the outer ring than a cross over to the min node on the inner ring is made, then following the inner ring (in the proper direction) till the max node of the inner ring, then the ring is closed by the

startnode (the max node of the outer ring). For the second face the route is exactly the other way around: from the min node on the outer ring following the outer ring (in the proper direction) till the max node on the outer ring than a cross over to the max node on the inner ring is made, then following the inner ring (in the proper direction) till the min node of the inner ring, then the ring is closed by the its start node (the min node of the outer ring). When an inner ring and outer ring share a node, this node is used for splitting point (instead of the max or the min node).

Scripts create structure

```
1  //CREATE STRUCTURE//
2
3
4  CREATE TABLE NODE (NODE_ID NUMBER NOT NULL, X NUMBER NOT NULL, Y NUMBER
   NOT NULL, Z NUMBER NOT NULL, NODE_IN_FACE_ID_REF NUMBER
5  CONSTRAINT NODE_PK PRIMARY KEY (NODE_ID)ENABLE);
6
7  CREATE TABLE EDGE (EDGE_ID NUMBER NOT NULL, STARTNODE NUMBER NOT NULL,
   ENDNODE NUMBER NOT NULL, EDGE_IN_FACE_ID_REF NUMBER
8  CONSTRAINT EDGE_PK PRIMARY KEY (EDGE_ID) ENABLE);
9
10 CREATE TABLE RING2EDGE (EDGE_ID_REF NUMBER NOT NULL, RING_ID_REF NUMBER
   NOT NULL, ORIENTATION VARCHAR2(1) NOT NULL,
11 CONSTRAINT RING2EDGE_PK PRIMARY KEY(EDGE_ID_REF, RING_ID_REF)ENABLE );
12
13 CREATE TABLE RING (RING_ID NUMBER NOT NULL, FACE_ID_REF NUMBER NOT NULL,
   INOUT VARCHAR2(1) NOT NULL, RING_ORDINATES MDSYS.SDO_ORDINATE_ARRAY,
14 CONSTRAINT RING_PK PRIMARY KEY (RING_ID) ENABLE);
15
16 CREATE TABLE FACE (FACE_ID NUMBER NOT NULL, SHELL_ID_REF_POS NUMBER NOT
   NULL, SHELL_ID_REF_NEG NUMBER NOT NULL,
17 CONSTRAINT FACE_PK PRIMARY KEY (FACE_ID) ENABLE);
18
19 CREATE TABLE SHELL (SHELL_ID NUMBER NOT NULL, VOLUME_ID_REF NUMBER NOT
   NULL, INOUT VARCHAR2(1) NOT NULL,
20 CONSTRAINT SHELL_PK PRIMARY KEY (SHELL_ID)ENABLE);
21
22 CREATE TABLE VOLUME (VOLUME_ID NUMBER NOT NULL,
23 CONSTRAINT VOLUME_PK PRIMARY KEY (VOLUME_ID)ENABLE);
24
25 ALTER TABLE EDGE ADD CONSTRAINT ENDNODE_FK FOREIGN KEY (ENDNODE)
   REFERENCES NODE (NODE_ID) ENABLE;
26 ALTER TABLE EDGE ADD CONSTRAINT STARTNODE_FK FOREIGN KEY (STARTNODE)
   REFERENCES NODE (NODE_ID) ENABLE;
27 ALTER TABLE RING2EDGE ADD CONSTRAINT EDGE_FK FOREIGN KEY (EDGE_ID_REF)
   REFERENCES EDGE (EDGE_ID) ENABLE;
28 ALTER TABLE RING2EDGE ADD CONSTRAINT RING_FK FOREIGN KEY (RING_ID_REF)
   REFERENCES RING (RING_ID) ENABLE;
29 ALTER TABLE RING ADD CONSTRAINT FACE_FK FOREIGN KEY (FACE_ID_REF)
   REFERENCES FACE (FACE_ID) ENABLE;
30 ALTER TABLE FACE ADD CONSTRAINT SHELLPOS_FK FOREIGN KEY
   (SHELL_ID_REF_POS) REFERENCES SHELL (SHELL_ID) ENABLE;
31 ALTER TABLE FACE ADD CONSTRAINT SHELLNEG_FK FOREIGN KEY
   (SHELL_ID_REF_NEG) REFERENCES SHELL (SHELL_ID) ENABLE;
32 ALTER TABLE SHELL ADD CONSTRAINT VOLUME_FK FOREIGN KEY (VOLUME_ID_REF)
   REFERENCES VOLUME (VOLUME_ID) ENABLE;
```

```
1  //CREATE VIEW NODE2SHELL//
2
3
4  --DROP VIEW node2shell;
5  CREATE VIEW node2shell AS
6
7  SELECT FACE.shell_id_ref_pos AS shell, RING.face_id_ref AS
   face,RING.ring_id AS ring,RING2EDGE.edge_id_ref AS edge, EDGE.startnode
   AS node FROM EDGE,RING2EDGE,RING,FACE
8  WHERE FACE.face_id=RING.face_id_ref AND RING.ring_id=
   RING2EDGE.ring_id_ref AND RING2EDGE.edge_id_ref=EDGE.edge_id
9  UNION ALL
10 SELECT FACE.shell_id_ref_pos AS shell, RING.face_id_ref AS
   face,RING.ring_id AS ring,RING2EDGE.edge_id_ref AS edge, EDGE.endnode AS
   node FROM EDGE,RING2EDGE,RING,FACE
11 WHERE FACE.face_id=RING.face_id_ref AND RING.ring_id=
   RING2EDGE.ring_id_ref AND RING2EDGE.edge_id_ref=EDGE.edge_id
12
13 UNION ALL
14
15 SELECT FACE.shell_id_ref_neg AS shell, RING.face_id_ref AS
   face,RING.ring_id AS ring,RING2EDGE.edge_id_ref AS edge,EDGE.startnode AS
   node FROM EDGE,RING2EDGE,RING,FACE
16 WHERE FACE.face_id=RING.face_id_ref AND RING.ring_id=
   RING2EDGE.ring_id_ref AND RING2EDGE.edge_id_ref=EDGE.edge_id
17 UNION ALL
18 SELECT FACE.shell_id_ref_neg AS shell, RING.face_id_ref AS
   face,RING.ring_id AS ring,RING2EDGE.edge_id_ref AS edge,EDGE.endnode AS
   node FROM EDGE,RING2EDGE,RING,FACE
19 WHERE FACE.face_id=RING.face_id_ref AND RING.ring_id=
   RING2EDGE.ring_id_ref AND RING2EDGE.edge_id_ref=EDGE.edge_id
20
21 ORDER BY shell,face,ring,edge,node;
```

```
1 //CREATE VIEW RINGORDINATES//
2
3
4 --DROP VIEW ringordinates;
5 CREATE VIEW ringordinates AS
6 SELECT ring_id, getRINGORDINATES(ring_id) as ringordinates FROM ring;
```

```
1  //CREATE INTERACTION TABLE FOR THE BENEFIT OF SDO_ANYINTERACT FUNCTION,
   WHICH IS USED IN VALIDATION TESTS//
2
3
4  --DROP TABLE chk_interaction CASCADE CONSTRAINTS;
5  CREATE TABLE chk_interaction (id NUMBER, geom SDO_GEOMETRY);
6
7  --DELETE FROM user_sdo_geom_metadata WHERE table_name = 'chk_interaction';
8  INSERT INTO user_sdo_geom_metadata (TABLE_NAME,COLUMN_NAME,DIMINFO,SRID)
9  VALUES ('chk_interaction','geom',
10 SDO_DIM_ARRAY(
11 SDO_DIM_ELEMENT('X', 84935, 86085, 0.005),
12 SDO_DIM_ELEMENT('Y', 444960, 446801, 0.005),
13 SDO_DIM_ELEMENT('Z', -5, 95, 0.005)
14 ),NULL);
15
16 --DROP INDEX SDX_interaction;
17 CREATE INDEX SDX_interaction ON chk_interaction(geom)
18 INDEXTYPE IS MDSYS.SPATIAL_INDEX
19 PARAMETERS ('SDO_INDX_DIMS=3');
```

Scripts validation tests

```
1  //VALIDATION TEST 1) UNIQUE PRIMITIVES//
2
3
4  SET SERVEROUTPUT ON
5
6  DECLARE
7  TYPE xyz_rec IS RECORD (x NUMBER, y NUMBER, z NUMBER);
8  TYPE xyz_type IS TABLE OF xyz_rec;
9  TYPE edge_rec1 IS RECORD (edge NUMBER, startnode NUMBER, endnode NUMBER);
10 TYPE edge_rec2 IS RECORD (startnode NUMBER, endnode NUMBER);
11 TYPE edge_type1 IS TABLE OF edge_rec1;
12 TYPE edge_type2 IS TABLE OF edge_rec2;
13 TYPE face_rec IS RECORD (rings NUMBER, min NUMBER, max NUMBER, edges
   NUMBER);
14 TYPE face_type IS TABLE OF face_rec;
15 TYPE element_type IS TABLE OF NUMBER;
16 tab_xyz xyz_type;
17 tab_node element_type;
18 tab_edge1 edge_type1;
19 tab_edge2 edge_type2;
20 tab_edge element_type;
21 tab_face face_type;
22 tab_face2 element_type;
23 tab_testface1 element_type;
24 tab_testface2 element_type;
25 vx NUMBER;
26 vy NUMBER;
27 vz NUMBER;
28 v_node NUMBER;
29 v_edge NUMBER;
30 v_startnode NUMBER;
31 v_endnode NUMBER;
32 v_result NUMBER;
33 result1 NUMBER:=0;
34 result2 NUMBER:=0;
35 result3 NUMBER:=0;
36 v_testface1 NUMBER;
37 v_testface2 NUMBER;
38 v_rings NUMBER;
39 v_min NUMBER;
40 v_max NUMBER;
41 v_edges NUMBER;
42
43
44 BEGIN
45
46 --check unique xyz-values of nodes
47 SELECT x,y,z BULK COLLECT INTO tab_xyz FROM node GROUP BY x,y,z HAVING
   COUNT(*)>1;
48 IF tab_xyz.count=0 THEN dbms_output.put_line('unique nodes');
49 ELSE
50 FOR i IN tab_xyz.FIRST..tab_xyz.LAST LOOP
51 vx:=tab_xyz(i).x;
52 vy:=tab_xyz(i).y;
53 vz:=tab_xyz(i).z;
54 dbms_output.put_line('nodes with equal coordinates
   ('||vx||','||vy||','||vz||'):');
55 SELECT node_id BULK COLLECT INTO tab_node FROM node WHERE x=vx AND y=vy
   AND z=vz;
56 FOR i IN tab_node.FIRST..tab_node.LAST LOOP
57 v_node:=tab_node(i);
58 dbms_output.put_line(v_node);
59 END LOOP;
```

```
60 END LOOP;
61 END IF;
62
63 --search for edges with same start- and end node (start node equals end
   node)
64 SELECT edge_id,startnode,endnode BULK COLLECT INTO tab_edge1 FROM edge
   where startnode= endnode;
65 IF tab_edge1.count=0 THEN result1:=1;
66 ELSE
67 FOR i IN tab_edge1.FIRST..tab_edge1.LAST LOOP
68 v_edge:=tab_edge1(i).edge;
69 dbms_output.put_line('edge '||v_edge||' (start node equals end node)');
70 END LOOP;
71 END IF;
72
73 --check edges with same begin- and end node
74 SELECT startnode,endnode BULK COLLECT INTO tab_edge2 FROM edge GROUP BY
   startnode,endnode HAVING COUNT(*)>1;
75 IF tab_edge2.count=0 THEN result2:=1;
76 ELSE FOR i IN tab_edge2.FIRST..tab_edge2.LAST LOOP
77 v_startnode:=tab_edge2(i).startnode;
78 v_endnode:=tab_edge2(i).endnode;
79 SELECT edge_id BULK COLLECT INTO tab_edge FROM edge WHERE
   startnode=v_startnode AND endnode=v_endnode;
80 FOR i IN tab_edge.FIRST..tab_edge.LAST LOOP
81 v_edge:=tab_edge(i);
82 dbms_output.put_line('edges with same start- and end node: '||v_edge);
83 END LOOP;
84 END LOOP;
85 END IF;
86
87 --search for edges where end node is begin node and begin node is end node
88 SELECT a.edge_id, a.startnode, a.endnode BULK COLLECT INTO tab_edge1 FROM
   edge a, edge b WHERE a.endnode= b.startnode AND a.startnode= b.endnode
   AND a.startnode<> b.startnode;
89 IF tab_edge1.count=0 THEN result3:=1;
90 ELSE
91 FOR i IN tab_edge1.FIRST..tab_edge1.LAST LOOP
92 v_edge:=tab_edge1(i).edge;
93 dbms_output.put_line('edge '||v_edge||' (end node = start node and start
   node = end node)');
94 END LOOP;
95 END IF;
96
97 --conclusion edges
98 v_result:=result1+result2+result3;
99 IF v_result=3 THEN dbms_output.put_line('unique edges'); END IF;
100
101 --check faces: first eliminate potentially equal faces based on some main
   characteristics
102 SELECT rings,min,max,edges BULK COLLECT INTO tab_face FROM (SELECT
   ring.face_id_ref AS face, count(distinct ring.ring_id) AS rings,
   min(ring2edge.edge_id_ref) AS min ,max(ring2edge.edge_id_ref) AS max,
   count (ring2edge.edge_id_ref) AS edges FROM ring2edge, ring WHERE
   ring.ring_id= ring2edge.ring_id_ref GROUP BY ring.face_id_ref) GROUP BY
   rings,min,max,edges HAVING COUNT(*)=2;
103 IF tab_face.count=0 THEN dbms_output.put_line('unique faces');
104
105 --then compare collections of edges of two faces with the same main
   characteristics
106 ELSE
107 FOR i IN tab_face.FIRST..tab_face.LAST LOOP
108 v_rings:=tab_face(i).rings;
```

```
109 v_min:=tab_face(i).min;
110 v_max:=tab_face(i).max;
111 v_edges:=tab_face(i).edges;
112 SELECT min(face) INTO v_testface1 FROM (SELECT ring.face_id_ref AS face,
    count(distinct ring.ring_id) AS rings, min(ring2edge.edge_id_ref) AS min
    ,max(ring2edge.edge_id_ref) AS max, count(ring2edge.edge_id_ref) AS edges
    FROM ring2edge, ring WHERE ring.ring_id= ring2edge.ring_id_ref GROUP BY
    ring.face_id_ref) WHERE rings=v_rings AND min=v_min AND max=v_max AND
    edges=v_edges;
113 SELECT max(face) INTO v_testface2 FROM (SELECT ring.face_id_ref AS face,
    count(distinct ring.ring_id) AS rings, min(ring2edge.edge_id_ref) AS min
    ,max(ring2edge.edge_id_ref) AS max, count(ring2edge.edge_id_ref) AS edges
    FROM ring2edge, ring WHERE ring.ring_id= ring2edge.ring_id_ref GROUP BY
    ring.face_id_ref) WHERE rings=v_rings AND min=v_min AND max=v_max AND
    edges=v_edges;
114 SELECT edge_id_ref BULK COLLECT INTO tab_testface1 FROM ring2edge,ring
    WHERE ring2edge.ring_id_ref=ring.ring_id AND ring.face_id_ref=v_testface1;
115 SELECT edge_id_ref BULK COLLECT INTO tab_testface2 FROM ring2edge,ring
    WHERE ring2edge.ring_id_ref=ring.ring_id AND ring.face_id_ref=v_testface2;
116 IF tab_testface1=tab_testface2 THEN dbms_output.put_line('equal faces:
    '||v_testface1||','||v_testface2); END IF;
117 END LOOP;
118 END IF;
119
120 --select not tested faces
121 SELECT rings,min,max,edges BULK COLLECT INTO tab_face FROM (SELECT
    ring.face_id_ref AS face, count(distinct ring.ring_id) AS rings,
    min(ring2edge.edge_id_ref) AS min ,max(ring2edge.edge_id_ref) AS max,
    count(ring2edge.edge_id_ref) AS edges FROM ring2edge, ring WHERE
    ring.ring_id= ring2edge.ring_id_ref GROUP BY ring.face_id_ref) GROUP BY
    rings,min,max,edges HAVING COUNT(*)>2;
122 IF tab_face.count>0 THEN dbms_output.put_line('faces not tested:');
123 FOR i IN tab_face.FIRST..tab_face.LAST LOOP
124 v_rings:=tab_face(i).rings;
125 v_min:=tab_face(i).min;
126 v_max:=tab_face(i).max;
127 v_edges:=tab_face(i).edges;
128 SELECT face BULK COLLECT INTO tab_face2 FROM (SELECT ring.face_id_ref AS
    face, count(distinct ring.ring_id) AS rings, min(ring2edge.edge_id_ref)
    AS min ,max(ring2edge.edge_id_ref) AS max, count(ring2edge.edge_id_ref)
    AS edges FROM ring2edge, ring WHERE ring.ring_id= ring2edge.ring_id_ref
    GROUP BY ring.face_id_ref) WHERE rings=v_rings AND min=v_min AND
    max=v_max AND edges=v_edges;
129 FOR i IN tab_face2.FIRST..tab_face2.LAST LOOP
130 dbms_output.put_line(tab_face2(i));
131 END LOOP;
132 END LOOP;
133 END IF;
134
135
136 END;
```

```
1  //VALIDATIONTEST 2) PRIMITIVE REFERENCES//
2
3
4  SET SERVEROUTPUT ON
5
6  DECLARE
7  TYPE volume_face_rec IS RECORD (volume NUMBER, face NUMBER);
8  TYPE ring_edge_rec IS RECORD (ring NUMBER, edges NUMBER);
9  TYPE face_edge_rec IS RECORD (face NUMBER, edge NUMBER);
10 TYPE volume_face_type IS TABLE OF volume_face_rec;
11 TYPE ring_edge_type IS TABLE OF ring_edge_rec;
12 TYPE face_edge_type IS TABLE OF face_edge_rec;
13 TYPE element_type IS TABLE OF NUMBER;
14 tab_face_edge FACE_EDGE_TYPE;
15 tab_ring_edge RING_EDGE_TYPE;
16 tab_volume_face VOLUME_FACE_TYPE;
17 tab_node ELEMENT_TYPE;
18 tab_edge ELEMENT_TYPE;
19 tab_ring ELEMENT_TYPE;
20 tab_face ELEMENT_TYPE;
21 tab_shell ELEMENT_TYPE;
22 v_node NUMBER;
23 v_edge NUMBER;
24 v_ring NUMBER;
25 v_face NUMBER;
26 v_shell NUMBER;
27 v_volume NUMBER;
28
29
30 BEGIN
31
32 --chk for isolated nodes
33 SELECT node_id BULK COLLECT INTO tab_node FROM node WHERE node_id NOT IN
   (SELECT startnode FROM edge) AND node_id NOT IN (SELECT endnode FROM edge);
34 IF tab_node.count=0 THEN dbms_output.put_line('no isolated nodes');
35 ELSE
36 dbms_output.put_line('isolated nodes:');
37 FOR i IN tab_node.FIRST..tab_node.LAST LOOP
38 v_node:=tab_node(i);
39 dbms_output.put_line(v_node);
40 END LOOP;
41 END IF;
42
43 --chk for isolated edges
44 SELECT edge_id BULK COLLECT INTO tab_edge FROM edge WHERE edge_id NOT IN
   (SELECT edge_id_ref FROM ring2edge);
45 IF tab_edge.count=0 THEN dbms_output.put_line('no isolated edges');
46 ELSE
47 dbms_output.put_line('isolated edges:');
48 FOR i IN tab_edge.FIRST..tab_edge.LAST LOOP
49 v_edge:=tab_edge(i);
50 dbms_output.put_line(v_edge);
51 END LOOP;
52 END IF;
53
54 --chk for nonexistent rings
55 SELECT DISTINCT ring_id BULK COLLECT INTO tab_ring FROM ring WHERE
   ring_id NOT IN (SELECT ring_id_ref FROM ring2edge);
56 IF tab_ring.count=0 THEN dbms_output.put_line('no nonexistent rings');
57 ELSE
58 dbms_output.put_line('nonexistent rings:');
59 FOR i IN tab_ring.FIRST..tab_ring.LAST LOOP
60 v_ring:=tab_ring(i);
```

```
61 dbms_output.put_line(v_ring);
62 END LOOP;
63 END IF;
64
65 --chk for nonexistent faces
66 SELECT DISTINCT face_id BULK COLLECT INTO tab_face FROM FACE WHERE
   face_id NOT IN (SELECT face_id_ref FROM ring);
67 IF tab_face.count=0 THEN dbms_output.put_line('no nonexistent faces');
68 ELSE
69 dbms_output.put_line('nonexistent faces:');
70 FOR i IN tab_face.FIRST..tab_face.LAST LOOP
71 v_face:=tab_face(i);
72 dbms_output.put_line(v_face);
73 END LOOP;
74 END IF;
75
76 --chk for nonexistent shells
77 SELECT DISTINCT shell_id BULK COLLECT INTO tab_shell FROM SHELL WHERE
   shell_id NOT IN (SELECT shell_id_ref_pos FROM FACE) AND shell_id NOT IN
   (SELECT shell_id_ref_neg FROM FACE);
78 IF tab_shell.count=0 THEN dbms_output.put_line('no nonexistent shells');
79 ELSE
80 dbms_output.put_line('nonexistent shells:');
81 FOR i IN tab_shell.FIRST..tab_shell.LAST LOOP
82 v_shell:=tab_shell(i);
83 dbms_output.put_line(v_shell);
84 END LOOP;
85 END IF;
86
87 --check for rings existing of less than 3 edges
88 SELECT ring_id_ref, count(DISTINCT edge_id_ref) BULK COLLECT INTO
   tab_ring_edge FROM ring2edge GROUP BY ring_id_ref HAVING COUNT(*) <3;
89 IF tab_ring_edge.count=0 THEN dbms_output.put_line('no rings with less
   than 3 edges');
90 ELSE
91 dbms_output.put_line('rings with less than 3 edges:');
92 FOR i IN tab_ring_edge.FIRST..tab_ring_edge.LAST LOOP
93 v_ring:=tab_ring_edge(i).ring;
94 dbms_output.put_line(v_ring);
95 END LOOP;
96 END IF;
97
98 --check for faces with rings (inner and outer rings) referring to the
   same edge
99 SELECT r.face_id_ref, r2e.edge_id_ref BULK COLLECT INTO tab_face_edge
   FROM ring2edge r2e,ring r WHERE r2e.ring_id_ref=r.ring_id GROUP BY
   r.face_id_ref, r2e.edge_id_ref HAVING COUNT(*)>1;
100 IF tab_face_edge.count=0 THEN dbms_output.put_line('no faces with inner
    and outer rings referring to the same edge');
101 ELSE
102 dbms_output.put_line('faces with rings referring to the same edges');
103 FOR i IN tab_face_edge.FIRST..tab_face_edge.LAST LOOP
104 v_face:=tab_face_edge(i).face;
105 v_edge:=tab_face_edge(i).edge;
106 dbms_output.put_line(v_face||','||v_edge);
107 END LOOP;
108 END IF;
109
110 --check for more than 1 node in a face which is used more than one time
111 SELECT face_id BULK COLLECT INTO tab_face FROM FACE;
112 FOR i IN tab_face.FIRST..tab_face.LAST LOOP
113 v_face:=tab_face(i);
114 select node BULK COLLECT INTO tab_node from (select startnode as node
```

```
    from edge where edge_id in (select edge from edge2shell where face=v_face)
115 union all
116 select endnode as node from edge where edge_id in (select edge from
    edge2shell where face=1)) group by node having count(*)>2;
117 IF tab_node.count>1 THEN dbms_output.put_line('more than one node is used
    two times in a face: '||v_face); END IF;
118 END LOOP;
119
120 --check for shells existing of less than 4 faces
121 SELECT shell BULK COLLECT INTO tab_shell FROM (SELECT shell_id_ref_pos AS
    shell, count(*) AS cnt FROM FACE GROUP BY shell_id_ref_pos
122 UNION ALL
123 SELECT shell_id_ref_neg AS shell, count(*) AS cnt FROM FACE GROUP BY
    shell_id_ref_neg) GROUP BY shell HAVING SUM(cnt)<4;
124 IF tab_shell.count=0 THEN dbms_output.put_line('no shells with less than
    4 faces');
125 ELSE
126 dbms_output.put_line('shells with less than 4 faces:');
127 FOR i IN tab_shell.FIRST..tab_shell.LAST LOOP
128 v_shell:=tab_shell(i);
129 dbms_output.put_line(v_shell);
130 END LOOP;
131 END IF;
132
133 --check for volumes with shells (inner and outer shells) referring to the
    same face
134 SELECT s2v.volume_id_ref, f2s.face_id BULK COLLECT INTO tab_volume_face
    FROM FACE f2s, SHELL s2v WHERE f2s.shell_id_ref_pos= s2v.shell_id OR
    f2s.shell_id_ref_neg= s2v.shell_id GROUP BY f2s.face_id,
    s2v.volume_id_ref HAVING COUNT(*)>1;
135 IF tab_volume_face.count=0 THEN dbms_output.put_line('no volumes with
    inner and outer shells referring to the same face');
136 ELSE
137 dbms_output.put_line('volumes with shells referring to the same face:');
138 FOR i IN tab_volume_face.FIRST..tab_volume_face.LAST LOOP
139 v_volume:=tab_volume_face(i).volume;
140 v_face:=tab_volume_face(i).face;
141 dbms_output.put_line(v_volume||','||v_face);
142 END LOOP;
143 END IF;
144
145 END;
```

```
1  //VALIDATIONTEST 3) ONE OUTER BOUNDARY//
2
3
4  SET SERVEROUTPUT ON
5
6  DECLARE
7  TYPE element_type IS TABLE OF NUMBER;
8  tab_face ELEMENT_TYPE;
9  tab_volume ELEMENT_TYPE;
10 v_face NUMBER;
11 v_volume NUMBER;
12
13
14 BEGIN
15
16 --chk for faces with too many (more than 1) outer rings
17 SELECT face_id_ref BULK COLLECT INTO tab_face FROM ring WHERE InOut='O'
   GROUP BY face_id_ref, InOut HAVING COUNT(*)>1;
18 IF tab_face.count=0 THEN dbms_output.put_line('no faces with more than 1
   outer ring');
19 ELSE
20 dbms_output.put_line('faces with more than 1 outer ring:');
21 FOR i IN tab_face.FIRST..tab_face.LAST LOOP
22 v_face:=tab_face(i);
23 dbms_output.put_line(v_face);
24 END LOOP;
25 END IF;
26
27 --chk for faces with only inner rings (no outer ring)
28 SELECT DISTINCT face_id_ref BULK COLLECT INTO tab_face FROM ring WHERE
   InOut='I' AND face_id_ref NOT IN (SELECT face_id_ref FROM ring WHERE
   InOut='O');
29 IF tab_face.count=0 THEN dbms_output.put_line('no faces with only inner
   rings (no outer ring)');
30 ELSE
31 dbms_output.put_line('faces with only inner rings (no outer ring):');
32 FOR i IN tab_face.FIRST..tab_face.LAST LOOP
33 v_face:=tab_face(i);
34 dbms_output.put_line(v_face);
35 END LOOP;
36 END IF;
37
38 --chk for faces with more than one inner rings
39 SELECT count(*) BULK COLLECT INTO tab_face FROM ring WHERE InOut='O'
   group by face_id_ref having count(*)>1;
40 IF tab_face.count=0 THEN dbms_output.put_line('no faces with more than
   one inner ring');
41 ELSE
42 dbms_output.put_line('faces with more than one inner ring:');
43 FOR i IN tab_face.FIRST..tab_face.LAST LOOP
44 v_face:=tab_face(i);
45 dbms_output.put_line(v_face);
46 END LOOP;
47 END IF;
48
49
50 --chk for volumes with too many (more than 1) outer shells
51 SELECT volume_id_ref BULK COLLECT INTO tab_volume FROM SHELL WHERE
   InOut='O' GROUP BY volume_id_ref, InOut HAVING COUNT(*)>1;
52 IF tab_volume.count=0 THEN dbms_output.put_line('no volumes with more
   than 1 outer shell');
53 ELSE
54 dbms_output.put_line('volumes with more than 1 outer shell:');
```

```
55 FOR i IN tab_volume.FIRST..tab_volume.LAST LOOP
56 v_volume:=tab_volume(i);
57 dbms_output.put_line(v_volume);
58 END LOOP;
59 END IF;
60
61 --chk for volumes with only inner shells (no outer shell), except for the
   universal volume
62 SELECT DISTINCT volume_id_ref BULK COLLECT INTO tab_volume FROM SHELL
   WHERE InOut='I' AND volume_id_ref NOT IN (SELECT volume_id_ref FROM SHELL
   WHERE InOut='O') AND volume_id_ref <>0;
63 IF tab_volume.COUNT=0 THEN dbms_output.put_line('no volumes with only
   inner shells (no outer shell)');
64 ELSE
65 dbms_output.put_line('volumes with only innner shells (no outer shell):');
66 FOR i IN tab_volume.FIRST..tab_volume.LAST LOOP
67 v_volume:=tab_volume(i);
68 dbms_output.put_line(v_volume);
69 END LOOP;
70 END IF;
71
72 --does the universal volume exist?
73 select volume_id bulk collect into tab_volume from volume where
   volume_id=0;
74 IF tab_volume.count=0 THEN dbms_output.put_line('no universal volume
   present'); end if;
75 select count(*) bulk collect into tab_volume from SHELL where inout='I'
   and volume_id_ref=0 group by volume_id_ref;
76 IF tab_volume.count=0 THEN dbms_output.put_line('the universal volume has
   not inner shells');end if;
77 select count(*) bulk collect into tab_volume from SHELL where inout='O'
   and volume_id_ref=0 group by volume_id_ref;
78 IF tab_volume.count>0 THEN dbms_output.put_line('the universal volume has
   a outer shell');end if;
79 END;
```

```
1  //VALIDATIONTEST 4) CLOSED RING//
2
3
4  SET SERVEROUTPUT ON
5
6  DECLARE
7  TYPE element_type IS TABLE OF NUMBER;
8  tab_ring ELEMENT_TYPE;
9  v_ring NUMBER;
10 v_ringordinates SDO_ORDINATE_ARRAY;
11
12
13 BEGIN
14
15 --select rings without valid ringordinates
16 SELECT ring_id BULK COLLECT INTO tab_ring FROM ringordinates WHERE
   ringordinates IS NULL;
17 IF tab_ring.count=0 THEN dbms_output.put_line('all rings are closed');
18 ELSE
19 FOR i IN tab_ring.FIRST..tab_ring.LAST LOOP
20 v_ring:=tab_ring(i);
21 dbms_output.put_line('ring without a closed boundary: '||v_ring);
22 SELECT getRINGORDINATES(ring_id) INTO v_ringordinates FROM ring WHERE
   ring_id=v_ring;
23 END LOOP;
24 END IF;
25
26 END;
```

```
1  //VALIDATIONTEST 4) CLOSED SHELL//
2
3
4  SET SERVEROUTPUT ON
5
6  DECLARE
7  TYPE element_type IS TABLE OF NUMBER;
8  tab_edge element_type;
9  tab_shell element_type;
10 v_shell NUMBER;
11 v_neg1 NUMBER;
12 v_neg2 NUMBER;
13 v_pos1 NUMBER;
14 v_pos2 NUMBER;
15 v_edge NUMBER;
16
17 BEGIN
18
19 --select all shells
20 SELECT shell_id BULK COLLECT INTO tab_shell FROM SHELL;
21 IF tab_shell.COUNT=0 THEN dbms_output.put_line('no shells in structure');
22 ELSE FOR i IN tab_shell.FIRST..tab_shell.LAST LOOP
23 v_shell:=tab_shell(i);
24
25 --select all edges from a shell
26 SELECT DISTINCT edge BULK COLLECT INTO tab_edge FROM node2shell WHERE
   shell=v_shell;
27
28 --check per collected edge for an even distribution of negative and
   positive references
29 IF tab_edge.count=0 THEN dbms_output.put_line('no edges in shell
   '||v_shell);
30 ELSE FOR i IN tab_edge.FIRST..tab_edge.LAST LOOP
31 v_edge:=tab_edge(i);
32
33 --select pos. oriented edge in neg. oriented face
34 SELECT count(*) INTO v_neg1 FROM ring2edge WHERE orientation='+' AND
   ring_id_ref IN (SELECT ring_id FROM ring WHERE face_id_ref IN (SELECT
   face_id FROM FACE WHERE shell_id_ref_neg=v_shell)) AND edge_id_ref=v_edge;
35 --dbms_output.put_line ('shell: '||v_shell||' edge: '||v_edge||' v_neg1:
   '||v_neg1);
36
37 --select pos. oriented edge in pos. oriented face
38 SELECT count(*) INTO v_pos1 FROM ring2edge WHERE orientation='+' AND
   ring_id_ref IN (SELECT ring_id FROM ring WHERE face_id_ref IN (SELECT
   face_id FROM FACE WHERE shell_id_ref_pos=v_shell)) AND edge_id_ref=v_edge;
39 --dbms_output.put_line ('shell: '||v_shell||' edge: '||v_edge||' v_pos1:
   '||v_pos1);
40
41 --select neg. oriented edge in pos. oriented face
42 SELECT count(*) INTO v_neg2 FROM ring2edge WHERE orientation='-' AND
   ring_id_ref IN (SELECT ring_id FROM ring WHERE face_id_ref IN (SELECT
   face_id FROM FACE WHERE shell_id_ref_pos=v_shell)) AND edge_id_ref=v_edge;
43 --dbms_output.put_line ('shell: '||v_shell||' edge: '||v_edge||' v_neg2:
   '||v_neg2);
44
45 --select neg. oriented edge in neg. oriented face
46 SELECT count(*) INTO v_pos2 FROM ring2edge WHERE orientation='-' AND
   ring_id_ref IN (SELECT ring_id FROM ring WHERE face_id_ref IN (SELECT
   face_id FROM FACE WHERE shell_id_ref_neg=v_shell)) AND edge_id_ref=v_edge;
47 --dbms_output.put_line ('shell: '||v_shell||' edge: '||v_edge||' v_pos2:
   '||v_pos2);
48
```

```
49 --conclusion
50 IF (v_neg1+v_neg2)<>(v_pos1+v_pos2) THEN dbms_output.put_line('no even
   distribution of edges in shell '||v_shell||', at edge '||v_edge); END IF;
51 END LOOP;
52 END IF;
53 END LOOP;
54 END IF;
55
56 END;
```

```
1  //VALIDATIONTEST 5) A PROPER FACE ORIENTATION//
2
3
4  SET SERVEROUTPUT ON
5
6  DECLARE
7  TYPE element_rec IS RECORD (face NUMBER, Iring NUMBER);
8  TYPE element_type IS TABLE OF element_rec;
9  tab_Irings element_type;
10 v_normalInner normal_array;
11 v_normalOuter normal_array;
12 v_face NUMBER;
13 v_Iring NUMBER;
14 v_Oring NUMBER;
15 v_resultX NUMBER;
16 v_resultY NUMBER;
17 v_resultZ NUMBER;
18
19 BEGIN
20
21 --select all inner rings
22 SELECT face_id_ref,ring_id BULK COLLECT INTO tab_Irings FROM ring WHERE
   InOut='I';
23
24 --compare normal of inner ring with normal of accompanying outer ring
25 IF tab_Irings.count=0 THEN dbms_output.put_line('no inner rings');
26 ELSE
27 dbms_output.put_line('inner rings in the following faces are not oriented
   properly (or not flat):');
28 FOR i IN tab_Irings.FIRST..tab_Irings.LAST LOOP
29 v_face:=tab_Irings(i).face;
30 v_Iring:=tab_Irings(i).Iring;
31 SELECT ring_id INTO v_Oring FROM ring WHERE face_id_ref=v_face AND
   InOut='O';
32 v_normalInner:=getNORMAL(v_Iring);
33 v_normalOuter:=getNORMAL(v_Oring);
34 v_resultX:=v_normalInner(1)+v_normalOuter(1);
35 v_resultX:=v_normalInner(2)+v_normalOuter(2);
36 v_resultX:=v_normalInner(3)+v_normalOuter(3);
37 IF v_resultX>0.01 OR v_resultY>0.01 OR v_resultZ>0.01 THEN
   dbms_output.put_line(v_face);END IF;
38 END LOOP;
39 END IF;
40
41 END;
```

```
1  //VALIDATIONTEST 5) A PROPER VOLUME ORIENTATION//
2
3
4  SET SERVEROUTPUT ON
5
6  DECLARE
7  TYPE element_type IS TABLE OF NUMBER;
8  tab_Ishell ELEMENT_TYPE;
9  tab_face ELEMENT_TYPE;
10 v_Ishell NUMBER;
11 v_face NUMBER;
12 v_Oring NUMBER;
13 v_minZ NUMBER;
14 v_shellNEG NUMBER;
15 v_shellPOS NUMBER;
16 v_normal normal_array;
17
18 BEGIN
19
20 --select all inner shells
21 SELECT shell_id BULK COLLECT INTO tab_Ishell FROM SHELL WHERE InOut='I';
22 FOR i IN tab_Ishell.FIRST..tab_Ishell.LAST LOOP
23 v_Ishell:=tab_Ishell(i);
24 --dbms_output.put_line('inner shell: '||v_Ishell);
25
26 SELECT MIN(z) INTO v_minZ FROM node WHERE node_id IN (SELECT node FROM
   node2shell WHERE shell=v_Ishell);
27 SELECT DISTINCT face BULK COLLECT INTO tab_face FROM node2shell,node
   WHERE shell=v_Ishell AND node.z=v_minZ AND node.node_id= node2shell.node;
28 --dbms_output.put_line(tab_face.count);
29 FOR i IN tab_face.FIRST..tab_face.LAST LOOP
30 v_face:=tab_face(i);
31 --dbms_output.put_line('face '||v_face);
32 SELECT ring_id INTO v_Oring FROM ring WHERE face_id_ref=v_face AND
   InOut='O';
33 --dbms_output.put_line ('outer ring '||v_Oring);
34 v_normal:=getNORMAL(v_Oring);
35 EXIT WHEN v_normal(3)<>0;
36 END LOOP;
37 --dbms_output.put_line('face '||v_face);
38 if v_normal(3)=0 THEN dbms_output.put_line('normalZ '||v_normal(3)); ENd
   if;
39 SELECT shell_id_ref_pos,shell_id_ref_neg INTO v_shellPOS, v_shellNEG FROM
   FACE WHERE face_id=v_face;
40 IF v_shellPOS=v_Ishell THEN IF v_normal(3)>=0 THEN
   dbms_output.put_line('1.not oriented properly'); END IF;
41 ELSIF v_shellNEG=v_Ishell THEN IF v_normal(3)>=0 THEN
   dbms_output.put_line('2.not oriented properly'); END IF;
42 END IF;
43
44 END LOOP;
45
46
47 END;
```

```
1  //VALIDATIONTEST 6) PLANAR FACES//
2
3
4  SET SERVEROUTPUT ON
5
6  DECLARE
7  v_tolerance NUMBER:=0.005;
8  TYPE element_type IS TABLE OF NUMBER;
9  tab_face element_type;
10 tab_Iring element_type;
11 v_face NUMBER;
12 v_Oring NUMBER;
13 v_Iring NUMBER;
14 x1 NUMBER;
15 y1 NUMBER;
16 z1 NUMBER;
17 x2 NUMBER;
18 y2 NUMBER;
19 z2 NUMBER;
20 x3 NUMBER;
21 y3 NUMBER;
22 z3 NUMBER;
23 x4 NUMBER;
24 y4 NUMBER;
25 z4 NUMBER;
26 a NUMBER:=4;
27 b NUMBER:=5;
28 c NUMBER:=6;
29 d NUMBER:=7;
30 e NUMBER:=8;
31 f NUMBER:=9;
32 vec1x NUMBER;
33 vec1y NUMBER;
34 vec1z NUMBER;
35 vec2x NUMBER;
36 vec2y NUMBER;
37 vec2z NUMBER;
38 normalX NUMBER;
39 normalY NUMBER;
40 normalZ NUMBER;
41 normalXouter NUMBER;
42 normalYouter NUMBER;
43 normalZouter NUMBER;
44 centroidX NUMBER;
45 centroidY NUMBER;
46 centroidZ NUMBER;
47 v_number NUMBER;
48 v_count NUMBER;
49 v_ordinate SDO_ORDINATE_ARRAY;
50 v_Nfactor NUMBER;
51 v_NfactorOuter NUMBER;
52 v_d NUMBER;
53 vx NUMBER;
54 vx_totaal NUMBER;
55 vy_totaal NUMBER;
56 vz_totaal NUMBER;
57 vy NUMBER;
58 vz NUMBER;
59 v_dOuter NUMBER;
60 Var_A NUMBER;
61 distance NUMBER;
62
63 BEGIN
```

```
64
65 --getting started
66 SELECT face_id BULK COLLECT INTO tab_face FROM FACE;
67 IF tab_face.count=0 THEN dbms_output.put_line('no faces');
68 ELSE FOR i IN tab_face.FIRST..tab_face.LAST LOOP
69 v_face:=tab_face(i);
70 dbms_output.put_line ('face: '||v_face);
71 SELECT ring_id INTO v_Oring FROM ring WHERE face_id_ref=v_face AND
   InOut='O';
72
73 --get centroid of outer ring
74 SELECT ringordinates INTO v_ordinate FROM ringordinates WHERE
   ring_id=v_Oring;
75 v_count:=v_ordinate.count;
76 v_count:=(v_count-3)/3;
77 --dbms_output.put_line('v_count '||v_count);
78 v_number:=1;
79 vx_totaal:=0;
80 vy_totaal:=0;
81 vz_totaal:=0;
82 FOR v_counter IN 1..v_count LOOP
83 --dbms_output.put_line('v_number '||v_number);
84 vx:=v_ordinate(v_number);
85 --dbms_output.put_line('vx '||vx);
86 vx_totaal:=(vx_totaal+vx);
87 --dbms_output.put_line('vx-totaal '||vx_totaal);
88 v_number:=v_number+1;
89 vy:=v_ordinate(v_number);
90 vy_totaal:=vy_totaal+vy;
91 v_number:=v_number+1;
92 vz:=v_ordinate(v_number);
93 vz_totaal:=vz+vz_totaal;
94 v_number:=v_number+1;
95 END LOOP;
96 CentroidX:=vx_totaal/v_count;
97 CentroidY:=vy_totaal/v_count;
98 CentroidZ:=vz_totaal/v_count;
99 --dbms_output.put_line('centroidX: '||CentroidX);
100 --dbms_output.put_line('centroidY: '||CentroidY);
101 --dbms_output.put_line('centroidZ: '||CentroidZ);
102
103 --select x,y,z values of four nodes (representing 2 edges)
104 x1:= v_ordinate(1);
105 y1:= v_ordinate(2);
106 z1:= v_ordinate(3);
107 x2:= v_ordinate(4);
108 y2:= v_ordinate(5);
109 z2:= v_ordinate(6);
110 x3:= v_ordinate(a);
111 y3:= v_ordinate(b);
112 z3:= v_ordinate(c);
113 x4:= v_ordinate(d);
114 y4:= v_ordinate(e);
115 z4:= v_ordinate(f);
116
117 --calculate the normal of the ring
118 vec1x:=x2-x1;
119 vec1y:=y2-y1;
120 vec1z:=z2-z1;
121 vec2x:=x4-x3;
122 vec2y:=y4-y3;
123 vec2z:=z4-z3;
124 normalX:= (vec1y * vec2z) - (vec1z * vec2y);
```

```
125 normalY:= -((vec2z * vec1x) - (vec2x * vec1z));
126 normalZ:= (vec1x * vec2y) - (vec1y * vec2x);
127
128 --in case the edges are parallel
129 IF normalX=0 AND normalY=0 AND normalZ=0 THEN
130 --select x,y,z values of two nodes representing another second edge
131 LOOP
132 SELECT ringordinates INTO v_ordinate FROM ringordinates WHERE
    ring_id=v_Oring;
133 dbms_output.put_line(v_Oring);
134 a:=a+1;
135 b:=b+1;
136 c:=c+1;
137 d:=d+1;
138 e:=e+1;
139 f:=f+1;
140 x3:= v_ordinate(a);
141 y3:= v_ordinate(b);
142 z3:= v_ordinate(c);
143 x4:= v_ordinate(d);
144 y4:= v_ordinate(e);
145 z4:= v_ordinate(f);
146 --calculate the normal of the ring
147 vec1x:=x2-x1;
148 vec1y:=y2-y1;
149 vec1z:=z2-z1;
150 vec2x:=x4-x3;
151 vec2y:=y4-y3;
152 vec2z:=z4-z3;
153 normalX:= (vec1y * vec2z) - (vec1z * vec2y);
154 normalY:= -((vec2z * vec1x) - (vec2x * vec1z));
155 normalZ:= (vec1x * vec2y) - (vec1y * vec2x);
156 --dbms_output.put_line(normalX||';'||normalY||';'||normalZ);
157 --dbms_output.put_line(normalX+normalY+normalZ);
158 Exit WHEN (normalX+normalY+normalZ)<>0;
159 END LOOP;
160 END IF;
161
162 --calculate distance to centroid
163 v_Nfactor:= sqrt((normalX*normalX)+(normalY*normalY)+(normalZ*normalZ));
164 --dbms_output.put_line('1) '||v_Nfactor);
165 v_d:=(normalX*-(x1))+(normalY*-(y1))+(normalZ*-(z1));
166 --dbms_output.put_line('2) '||v_d);
167 Var_A:=normalX*CentroidX+normalY*CentroidY+normalZ*CentroidZ+v_d;
168 --dbms_output.put_line('3) '||var_A);
169 distance:=Var_A/v_Nfactor;
170 --dbms_output.put_line('4) '||distance);
171 IF distance between -(v_tolerance) AND v_tolerance THEN
    dbms_output.enable; dbms_output.put_line('distance centroid to outer ring
    within tolerance value IS flat');
172 ELSE dbms_output.enable; dbms_output.put_line('distance centroid to outer
    ring greater than tolerance value IS NOT flat');
173 END IF;
174 --IF distance >-(v_tolerance) THEN dbms_output.put_line('distance
    centroid to outer ring of face('||v_face||'): '||distance); END IF;
175
176 --check for inner rings
177 SELECT ring_id BULK COLLECT INTO tab_Iring FROM ring WHERE
    face_id_ref=v_face AND InOut='I';
178 IF tab_Iring.count>0
179 THEN FOR i IN tab_Iring.FIRST..tab_Iring.LAST LOOP
180 v_Iring:=tab_Iring(i);
181 normalXouter:=normalX;
```

```
182 normalYouter:=normalY;
183 normalZouter:=normalZ;
184 v_dOuter:=v_d;
185 v_NfactorOuter:=v_Nfactor;
186
187
188 --chk centroid within inner ring
189 --get centroid of inner ring
190 SELECT ringordinates INTO v_ordinate FROM ringordinates WHERE
    ring_id=v_Iring;
191 v_count:=v_ordinate.count;
192 v_count:=(v_count-3)/3;
193 v_number:=1;
194 vx_totaal:=0;
195 vy_totaal:=0;
196 vz_totaal:=0;
197 FOR v_counter IN 1..v_count LOOP
198 vx:=v_ordinate(v_number);
199 vx_totaal:=(vx_totaal+vx);
200 v_number:=v_number+1;
201 vy:=v_ordinate(v_number);
202 vy_totaal:=(vy_totaal+vy);
203 v_number:=v_number+1;
204 vz:=v_ordinate(v_number);
205 vz_totaal:=(vz_totaal+vz);
206 v_number:=v_number+1;
207 END LOOP;
208 CentroidX:=vx_totaal/v_count;
209 CentroidY:=vy_totaal/v_count;
210 CentroidZ:=vz_totaal/v_count;
211
212 --select x,y,z values of four nodes (representing 2 edges)
213 x1:= v_ordinate(1);
214 y1:= v_ordinate(2);
215 z1:= v_ordinate(3);
216 x2:= v_ordinate(4);
217 y2:= v_ordinate(5);
218 z2:= v_ordinate(6);
219 x3:= v_ordinate(a);
220 y3:= v_ordinate(b);
221 z3:= v_ordinate(c);
222 x4:= v_ordinate(d);
223 y4:= v_ordinate(e);
224 z4:= v_ordinate(f);
225
226 --calculate the normal of the ring
227 vec1x:=x2-x1;
228 vec1y:=y2-y1;
229 vec1z:=z2-z1;
230 vec2x:=x4-x3;
231 vec2y:=y4-y3;
232 vec2z:=z4-z3;
233 normalX:= (vec1y * vec2z) - (vec1z * vec2y);
234 normalY:= -((vec2z * vec1x) - (vec2x * vec1z));
235 normalZ:= (vec1x * vec2y) - (vec1y * vec2x);
236
237 --in case the edges are parallel
238 IF normalX=0 AND normalY=0 AND normalZ=0 THEN
239 --select x,y,z values of two nodes representing another second edge
240 LOOP
241 SELECT ringordinates INTO v_ordinate FROM ringordinates WHERE
    ring_id=v_Iring;
242 a:=a+1;
```

```
243 b:=b+1;
244 c:=c+1;
245 d:=d+1;
246 e:=e+1;
247 f:=f+1;
248 x3:= v_ordinate(a);
249 y3:= v_ordinate(b);
250 z3:= v_ordinate(c);
251 x4:= v_ordinate(d);
252 y4:= v_ordinate(e);
253 z4:= v_ordinate(f);
254 --calculate the normal of the ring
255 vec1x:=x2-x1;
256 vec1y:=y2-y1;
257 vec1z:=z2-z1;
258 vec2x:=x4-x3;
259 vec2y:=y4-y3;
260 vec2z:=z4-z3;
261 normalX:= (vec1y * vec2z) - (vec1z * vec2y);
262 normalY:= -((vec2z * vec1x) - (vec2x * vec1z));
263 normalZ:= (vec1x * vec2y) - (vec1y * vec2x);
264 CONTINUE WHEN normalX=0 AND normalY=0 AND normalZ=0;
265 END LOOP;
266 END IF;
267
268 --calculate distance to centroid
269 v_Nfactor:= sqrt((normalX*normalX)+(normalY*normalY)+(normalZ*normalZ));
270 v_d:=(normalX*-(x1))+(normalY*-(y1))+(normalZ*-(z1));
271 Var_A:=normalX*CentroidX+normalY*CentroidY+normalZ*CentroidZ+v_d;
272 distance:=Var_A/v_Nfactor;
273 IF distance >v_tolerance THEN dbms_output.put_line('distance centroid to
    inner ring('||v_Iring||') of face('||v_face||'): '||distance); END IF;
274 IF distance <-(v_tolerance) THEN dbms_output.put_line('distance centroid
    to inner ring('||v_Iring||') of face('||v_face||'): '||distance); END IF;
275
276 --chk centroid with outer ring
277 --calculate distance of centroid (inner ring) to outer ring
278 Var_A:=normalXouter*CentroidX+normalYouter*CentroidY+normalZouter*CentroidZ
    +v_dOuter;
279 distance:=Var_A/v_NfactorOuter;
280
281 IF distance >v_tolerance THEN dbms_output.put_line('distance centroid of
    inner ring('||v_Iring||') to outer ring of face('||v_face||'):
    '||distance); END IF;
282 IF distance <-(v_tolerance) THEN dbms_output.put_line('distance centroid
    of inner ring('||v_Iring||') to outer ring of face('||v_face||'):
    '||distance); END IF;
283 END LOOP;
284 END IF;
285 END LOOP;
286 END IF;
287 END;
```

```
1  //VALIDATIONTEST 7) SELF INTERSECTING VOLUMES//
2
3
4  SET SERVEROUTPUT ON
5
6  DECLARE
7  TYPE element_type IS TABLE OF NUMBER;
8  tab_volume element_type;
9  tab_Ishell element_type;
10 tab_faces element_type;
11 tab_edges element_type;
12 tab_chkedges element_type;
13 tab_nodes1 element_type;
14 tab_nodes2 element_type;
15 tab_nodes3 element_type;
16 tab_check element_type;
17 v_volume NUMBER;
18 v_Ishell NUMBER;
19 v_Oshell NUMBER;
20 v_face NUMBER;
21 v_edge NUMBER;
22 v_startnode NUMBER;
23 v_endnode NUMBER;
24 v_test NUMBER;
25 vx NUMBER;
26 vy NUMBER;
27 vz NUMBER;
28 nr NUMBER;
29 pointtype SDO_POINT_TYPE;
30 v_ordinates SDO_ORDINATE_ARRAY;
31 v_point SDO_GEOMETRY;
32 v_shellgeom SDO_GEOMETRY;
33 v_fgeom SDO_GEOMETRY;
34 sql_truncate VARCHAR(100):='truncate table chk_interaction';
35
36
37 BEGIN
38
39
40 --getting started
41 execute immediate sql_truncate;
42 SELECT DISTINCT volume_id_ref BULK COLLECT INTO tab_volume FROM SHELL;
43 IF tab_volume.count=0 THEN dbms_output.put_line('no volumes in structure');
44 ELSE
45 FOR i IN tab_volume.FIRST..tab_volume.LAST LOOP
46 v_volume:=tab_volume(i);
47 dbms_output.put_line('volume: '||v_volume);
48
49 --select edges per volume
50 SELECT DISTINCT edge BULK COLLECT INTO tab_edges FROM node2shell WHERE
   shell IN (SELECT shell_id FROM SHELL WHERE volume_id_ref=v_volume);
51
52 --insert all edges in chk_interaction table and add geometry
53 nr:=tab_edges.count;
54 FORALL i IN 1..nr
55 INSERT INTO chk_interaction (id) VALUES (tab_edges(i));
56 UPDATE chk_interaction SET geom= getLINE(id);
57
58 --select all faces per volume
59 SELECT DISTINCT face_id BULK COLLECT INTO tab_faces FROM FACE WHERE
   shell_id_ref_pos IN (SELECT shell_id FROM SHELL WHERE
   volume_id_ref=v_volume) OR shell_id_ref_neg IN (SELECT shell_id FROM
   SHELL WHERE volume_id_ref=v_volume);
```

```
60 IF tab_faces.count=0 THEN dbms_output.put_line('no faces selected');
61
62 --chk interaction edges per face (where edge is not part of that face)
63 ELSE FOR i IN tab_faces.FIRST..tab_faces.LAST LOOP
64 v_face:=tab_faces(i);
65 --dbms_output.put_line('face: '||v_face);
66 v_fgeom:=getPOLYGON(v_face);
67 SELECT id BULK COLLECT INTO tab_chkedges FROM chk_interaction WHERE
   (SDO_ANYINTERACT(geom,v_fgeom)='TRUE') AND (id NOT IN (SELECT
   r2e.edge_id_ref FROM ring2edge r2e, ring r WHERE
   r2e.ring_id_ref=r.ring_id AND face_id_ref=v_face));
68
69 --chk edges which interact with the specified face for sharing nodes
70 IF tab_chkedges.count>0 THEN --dbms_output.put_line('closer check needed
   for face '||v_face);
71 FOR i IN tab_chkedges.FIRST..tab_chkedges.LAST LOOP
72 v_edge:=tab_chkedges(i);
73 --dbms_output.put_line(v_edge);
74 SELECT node BULK COLLECT INTO tab_nodes1 FROM (SELECT startnode AS node
   FROM edge WHERE edge_id=v_edge
75 UNION ALL SELECT endnode AS node FROM edge WHERE edge_id=v_edge);
76 SELECT DISTINCT node BULK COLLECT INTO tab_nodes2 FROM (
77 SELECT startnode AS node FROM edge WHERE edge_id IN (SELECT
   r2e.edge_id_ref FROM ring2edge r2e, ring r WHERE
   r2e.ring_id_ref=r.ring_id AND face_id_ref=v_face)
78 UNION ALL
79 SELECT endnode AS node FROM edge WHERE edge_id IN (SELECT r2e.edge_id_ref
   FROM ring2edge r2e, ring r WHERE r2e.ring_id_ref=r.ring_id AND
   face_id_ref=v_face));
80 tab_nodes3:=tab_nodes1 MULTISET INTERSECT tab_nodes2;
81 IF tab_nodes3.count=0 THEN dbms_output.put_line ('face '||v_face||' has a
   non-valid intersection with another face of the same volume
   ('||v_volume||')');
82 END IF;
83 END LOOP;
84 END IF;
85 END LOOP;
86 END IF;
87
88
89 --closure
90 execute immediate sql_truncate;
91 END LOOP;
92 END IF;
93
94 END;
```

```
1  //VALIDATIONTEST 7) EVERY INNER RING MUST BE INSIDE ITS ACCOMPANYING
   OUTER RING//
2
3
4  SET SERVEROUTPUT ON
5
6  DECLARE
7  TYPE element_type IS TABLE OF NUMBER;
8  tab_Iring element_type;
9  tab_edges element_type;
10 tab_edgesOUTER element_type;
11 tab_chkedges element_type;
12 tab_check element_type;
13 v_Iring NUMBER;
14 v_Oring NUMBER;
15 v_edge NUMBER;
16 nr NUMBER;
17 vx NUMBER;
18 vy NUMBER;
19 vz NUMBER;
20 v_linegeom SDO_GEOMETRY;
21 pointtype SDO_POINT_TYPE;
22 v_point SDO_GEOMETRY;
23 v_ringgeom SDO_GEOMETRY;
24 v_ordinates SDO_ORDINATE_ARRAY;
25 sql_truncate VARCHAR(100):='truncate table chk_interaction';
26
27
28 BEGIN
29
30 --getting started, select inner rings
31 execute immediate sql_truncate;
32 SELECT DISTINCT ring_id BULK COLLECT INTO tab_Iring FROM ring WHERE
   InOut='I';
33 IF tab_Iring.count=0 THEN dbms_output.put_line('no inner rings in
   structure');
34 ELSE
35 FOR i IN tab_Iring.FIRST..tab_Iring.LAST LOOP
36 v_Iring:=tab_Iring(i);
37 dbms_output.put_line('inner ring: '||v_Iring);
38
39
40 --select all edges of inner ring and insert them in the chk_interaction
   table and add geometry
41 SELECT DISTINCT edge BULK COLLECT INTO tab_edges FROM edge2shell WHERE
   ring=v_Iring;
42 nr:=tab_edges.count;
43 FORALL i IN 1..nr
44 INSERT INTO chk_interaction (id) VALUES (tab_edges(i));
45 UPDATE chk_interaction SET geom= getLINE(id);
46
47 --select accompanying outer ring
48 SELECT ring_id INTO v_Oring FROM ring WHERE InOut='O' AND face_id_ref IN
   (SELECT face_id_ref FROM ring WHERE InOut='I' AND ring_id=v_Iring);
49 SELECT ringordinates INTO v_ordinates FROM ringordinates WHERE
   ring_id=v_Oring;
50 v_ringgeom:=SDO_GEOMETRY(3003,NULL,NULL,SDO_ELEM_INFO_ARRAY(1,1003,1),v_ord
   inates);
51 SELECT id BULK COLLECT INTO tab_check FROM chk_interaction WHERE
   (SDO_ANYINTERACT(geom,v_ringgeom)='TRUE');
52 IF tab_check.count<>nr THEN dbms_output.put_line('inner ring not
   (completely) inside outer ring'); END IF;
53 END LOOP;
```

```
54
55
56 --closure
57 execute immediate sql_truncate;
58 END IF;
59
60
61 END;
```

```
1  //VALIDATIONTEST 7) EVERY INNER SHELL MUST BE INSIDE ITS ACCOMPANYING
   OUTER SHELL//
2
3
4  SET SERVEROUTPUT ON
5
6  DECLARE
7  TYPE element_type IS TABLE OF NUMBER;
8  tab_volume element_type;
9  tab_Ishell element_type;
10 tab_faces element_type;
11 tab_edges element_type;
12 tab_chkedges element_type;
13 tab_nodes1 element_type;
14 tab_nodes2 element_type;
15 tab_nodes3 element_type;
16 tab_check element_type;
17 v_volume NUMBER;
18 v_Ishell NUMBER;
19 v_Oshell NUMBER;
20 v_face NUMBER;
21 v_edge NUMBER;
22 v_startnode NUMBER;
23 v_endnode NUMBER;
24 v_test NUMBER;
25 vx NUMBER;
26 vy NUMBER;
27 vz NUMBER;
28 nr NUMBER;
29 pointtype SDO_POINT_TYPE;
30 v_ordinates SDO_ORDINATE_ARRAY;
31 v_point SDO_GEOMETRY;
32 v_shellgeom SDO_GEOMETRY;
33 v_sgeom SDO_GEOMETRY;
34 sql_truncate VARCHAR(100):='truncate table chk_interaction';
35
36
37 BEGIN
38
39
40 --getting started
41 execute immediate sql_truncate;
42
43
44 --select all inner shells
45 SELECT shell_id BULK COLLECT INTO tab_Ishell FROM SHELL WHERE InOut='I'
   AND volume_id_ref<>0;
46 IF tab_Ishell.COUNT=0 THEN dbms_output.put_line('no inner shells'); ELSE
47 FOR i IN tab_Ishell.FIRST..tab_Ishell.LAST LOOP
48 v_Ishell:=tab_Ishell(i);
49 dbms_output.put_line('inner shell: '||v_Ishell);
50
51 --select all faces of the inner shell and insert them in the
   chk_interaction table and add geometry
52 SELECT DISTINCT face_id BULK COLLECT INTO tab_faces FROM FACE WHERE
   shell_id_ref_pos=v_Ishell OR shell_id_ref_neg = v_Ishell;
53 --dbms_output.put_line(tab_faces.count);
54 IF tab_faces.count=0 THEN dbms_output.put_line('no faces selected');
55 ELSE
56 nr:=tab_faces.count;
57 FOR i IN tab_faces.FIRST..tab_faces.LAST LOOP
58 v_face:=tab_faces(i);
59 --dbms_output.put_line(v_face);
```

```
60 INSERT INTO chk_interaction (id) VALUES (v_face);
61 END LOOP;
62 UPDATE chk_interaction SET geom= getPOLYGON(id);
63
64 --select accompanying outer shell and chk interaction of the faces of the
   inner shell with the outer shell
65 SELECT shell_id INTO v_Oshell FROM SHELL WHERE InOut='O' AND
   volume_id_ref IN (SELECT volume_id_ref FROM SHELL WHERE InOut='I' AND
   shell_id=v_Ishell);
66 v_sgeom:=getSOLID(v_Oshell);
67 --dbms_output.put_line(v_Oshell);
68 SELECT id BULK COLLECT INTO tab_check FROM chk_interaction WHERE
   (SDO_ANYINTERACT(geom,v_sgeom)='TRUE');
69 IF tab_check.count<>nr THEN dbms_output.put_line('inner shell not
   (completely) inside outer shell');
70 --dbms_output.put_line(tab_check.count||';'||nr);
71 END IF;
72
73
74 --closure
75 execute immediate sql_truncate;
76
77 END IF;
78 END LOOP;
79 END IF;
80
81 END;
```

Scripts geometry operations

```
1  //OPERATION GETPOINT//
2
3
4  CREATE OR REPLACE FUNCTION getpoint(i_node IN NUMBER)
5  RETURN SDO_GEOMETRY IS
6
7  pointtype sdo_point_type;
8  vx NUMBER;
9  vy NUMBER;
10 vz NUMBER;
11 v_point SDO_GEOMETRY;
12 v_node NUMBER;
13
14 BEGIN
15
16 v_node:=i_node;
17 SELECT x INTO vx FROM node WHERE node_id=v_node;
18 SELECT y INTO vy FROM node WHERE node_id=v_node;
19 SELECT z INTO vz FROM node WHERE node_id=v_node;
20 pointtype:=sdo_point_type(vx,vy,vz);
21 v_point:=SDO_GEOMETRY (3001, NULL, pointtype, NULL,NULL);
22 RETURN v_point;
23
24 END;
```

```
1  //OPERATION GETLINE//
2
3
4  CREATE OR REPLACE FUNCTION getLINE(i_edge IN NUMBER)
5  RETURN SDO_GEOMETRY AS
6
7  v_startnode NUMBER;
8  v_endnode NUMBER;
9  v_edge NUMBER;
10 v_xs NUMBER;
11 v_ys NUMBER;
12 v_zs NUMBER;
13 v_xe NUMBER;
14 v_ye NUMBER;
15 v_ze NUMBER;
16 v_ordinate SDO_ORDINATE_ARRAY;
17 LINE SDO_GEOMETRY;
18
19 BEGIN
20
21 --getting started
22 v_edge:=i_edge;
23
24 --select begin and end node
25 SELECT startnode INTO v_startnode FROM edge WHERE edge_id=v_edge;
26 SELECT endnode INTO v_endnode FROM edge WHERE edge_id=v_edge;
27 SELECT x INTO v_xs FROM node WHERE node_id= v_startnode;
28 SELECT y INTO v_ys FROM node WHERE node_id= v_startnode;
29 SELECT z INTO v_zs FROM node WHERE node_id= v_startnode;
30 SELECT x INTO v_xe FROM node WHERE node_id= v_endnode;
31 SELECT y INTO v_ye FROM node WHERE node_id= v_endnode;
32 SELECT z INTO v_ze FROM node WHERE node_id= v_endnode;
33 v_ordinate:=sdo_ordinate_array(v_xs,v_ys,v_zs,v_xe,v_ye,v_ze);
34
35 --closure
36 IF v_ordinate.count=6
37 THEN
   LINE:=SDO_GEOMETRY(3002,NULL,NULL,SDO_ELEM_INFO_ARRAY(1,2,1),v_ordinate);
38 ELSE LINE:=NULL; END IF;
39 RETURN LINE;
40 EXCEPTION WHEN NO_DATA_FOUND THEN LINE:=NULL;
41 RETURN LINE;
42
43 END;
```

```
1  //OPERATION GETPOLYGON//
2
3
4  CREATE OR REPLACE FUNCTION getPOLYGON(i_face IN NUMBER)
5  RETURN SDO_GEOMETRY AS
6
7  TYPE ring_type IS TABLE OF NUMBER;
8  tab_Iring ring_type;
9  v_Oring NUMBER;
10 v_Iring NUMBER;
11 v_x NUMBER;
12 v_y NUMBER;
13 v_z NUMBER;
14 x NUMBER;
15 v_offset NUMBER;
16 v_count NUMBER;
17 v_ordinate SDO_ORDINATE_ARRAY;
18 v_Iordinate SDO_ORDINATE_ARRAY;
19 v_info_array SDO_ELEM_INFO_ARRAY;
20 FACE SDO_GEOMETRY;
21
22 BEGIN
23
24 --select outer ring and inner rings
25 SELECT ring_id INTO v_Oring FROM ring WHERE face_id_ref=i_face AND
   InOut='O';
26 SELECT ringordinates INTO v_ordinate FROM ringordinates WHERE
   ring_id=v_Oring;
27 SELECT ring_id BULK COLLECT INTO tab_Iring FROM ring WHERE
   face_id_ref=i_face AND InOut='I';
28 v_info_array:=sdo_elem_info_array(1,1003,1);
29
30 --when no innerrings present
31 IF tab_Iring.count=0 THEN
32 FACE:=SDO_GEOMETRY(3003,NULL,NULL,v_info_array, v_ordinate); RETURN FACE;
33
34 --when inner rings are present:
35 ELSE FOR i IN tab_Iring.FIRST..tab_Iring.LAST LOOP
36 v_Iring:=tab_Iring(i);
37 SELECT ringordinates INTO v_Iordinate FROM ringordinates WHERE
   ring_id=v_Iring;
38
39 --adjust array-info
40 v_offset:=v_ordinate.count+1;
41 v_count:=v_info_array.count+1;
42 v_info_array.extend;
43 v_info_array(v_count):=(v_offset);
44 v_count:=v_count+1;
45 v_info_array.extend;
46 v_info_array(v_count):=(2003);
47 v_count:=v_count+1;
48 v_info_array.extend;
49 v_info_array(v_count):=(1);
50
51 --extend ordinate-array
52 x:=v_Iordinate.count;
53 FOR v_counter IN 1..x LOOP
54 v_ordinate.extend;
55 v_ordinate(v_offset):=v_Iordinate(v_counter);
56 v_offset:=v_offset+1;
57 END LOOP;
58 END LOOP;
59
```

```
60 FACE:=SDO_GEOMETRY(3003,NULL,NULL,v_info_array,v_ordinate);
61 RETURN FACE;
62 END IF;
63 EXCEPTION WHEN NO_DATA_FOUND THEN FACE:=NULL;RETURN FACE;
64 RETURN FACE;
65
66 END;
```

```
1  //OPERATION GETSOLID//
2
3
4  CREATE OR REPLACE FUNCTION getSOLID(i_volume IN NUMBER)
5  RETURN SDO_GEOMETRY AS
6
7  TYPE element_type IS TABLE OF NUMBER;
8  tab_Ishell element_type;
9  tab_POSface element_type;
10 tab_NEGface element_type;
11 v_volume NUMBER;
12 v_Oshell NUMBER;
13 v_Ishell NUMBER;
14 v_face NUMBER;
15 v_nrfaceS NUMBER;
16 v_offset NUMBER;
17 v_count NUMBER;
18 v_loop NUMBER;
19 x NUMBER;
20 v_innerrings NUMBER;
21 v_face1ordinate SDO_ORDINATE_ARRAY;
22 v_face2ordinate SDO_ORDINATE_ARRAY;
23 v_ordinate SDO_ORDINATE_ARRAY;
24 v_Fordinate SDO_ORDINATE_ARRAY;
25 v_info_array SDO_ELEM_INFO_ARRAY;
26 VOLUME SDO_GEOMETRY;
27
28 BEGIN
29
30 --getting started
31 v_volume:=i_volume;
32 v_ordinate:=sdo_ordinate_array();
33 v_info_array:=sdo_elem_info_array(1,1007,1);
34
35 --select shells (outer and inner)
36 SELECT shell_id INTO v_Oshell FROM SHELL WHERE volume_id_ref=v_volume AND
   InOut='O';
37 SELECT shell_id BULK COLLECT INTO tab_Ishell FROM SHELL WHERE
   volume_id_ref=v_volume AND InOut='I';
38
39
40 --get faces of outer shell
41 SELECT face_id BULK COLLECT INTO tab_POSface FROM FACE WHERE
   shell_id_ref_pos=v_Oshell;
42 SELECT face_id BULK COLLECT INTO tab_NEGface FROM FACE WHERE
   shell_id_ref_neg=v_Oshell;
43 v_nrfaces:=tab_POSface.count+tab_NEGface.count;
44 --dbms_output.put_line('nr of faces: '||v_nrfaces);
45
46 --adjust array-info for outer shell
47 v_info_array.extend;
48 v_info_array(4):=1;
49 v_info_array.extend;
50 v_info_array(5):=1006;
51 v_info_array.extend;
52 v_info_array(6):=(v_nrfaces);
53
54 --get pos. oriented faces first (of outer shell)
55 IF tab_POSface.count>0 THEN
56 FOR i IN tab_POSface.FIRST..tab_POSface.LAST LOOP
57 v_face:=tab_POSface(i);
58 --dbms_output.put_line('(pos) face: '||v_face);
59
```

```
60  SELECT count(*) INTO v_innerrings FROM ring WHERE InOut='I' AND
    face_id_ref=v_face;
61  --dbms_output.put_line('count inner rings: '||v_innerrings);
62  IF v_innerrings>1 THEN dbms_output.put_line ('more than 1 inner ring in
    face: '||v_face||' (geen solid geconstrueerd'); VOLUME:=NULL;
63  ELSIF v_innerrings=1 THEN dbms_output.put_line ('1 inner ring, face:
    '||v_face);
64  v_nrfaces:=v_nrfaces+1;
65  --dbms_output.put_line ('number of faces: '||v_nrfaces);
66  v_info_array(6):=(v_nrfaces);
67  deleteINNERRINGS(v_face,v_face1ordinate,v_face2ordinate);
68
69  --adjust array-info for face1
70  v_offset:=v_ordinate.count+1;
71  v_count:=v_info_array.count+1;
72  v_info_array.extend;
73  v_info_array(v_count):=(v_offset);
74  v_count:=v_count+1;
75  v_info_array.extend;
76  v_info_array(v_count):=(1003);
77  v_count:=v_count+1;
78  v_info_array.extend;
79  v_info_array(v_count):=(1);
80  --extend ordinate-array for face1
81  x:=v_face1ordinate.count;
82  FOR v_counter IN 1..x LOOP
83  v_ordinate.extend;
84  v_ordinate(v_offset):=v_face1ordinate(v_counter);
85  v_offset:=v_offset+1;
86  END LOOP;
87
88  --adjust array-info for face2
89  v_offset:=v_ordinate.count+1;
90  v_count:=v_info_array.count+1;
91  v_info_array.extend;
92  v_info_array(v_count):=(v_offset);
93  v_count:=v_count+1;
94  v_info_array.extend;
95  v_info_array(v_count):=(1003);
96  v_count:=v_count+1;
97  v_info_array.extend;
98  v_info_array(v_count):=(1);
99  --extend ordinate-array for face2
100 x:=v_face2ordinate.count;
101 FOR v_counter IN 1..x LOOP
102 v_ordinate.extend;
103 v_ordinate(v_offset):=v_face2ordinate(v_counter);
104 v_offset:=v_offset+1;
105 END LOOP;
106
107 ELSE SELECT ringordinates INTO v_Fordinate FROM ringordinates WHERE
    ring_id IN (SELECT ring_id FROM ring WHERE face_id_ref=v_face AND
    InOut='O');
108 --dbms_output.put_line('geen inner ring');
109
110 --adjust array-info
111 v_offset:=v_ordinate.count+1;
112 v_count:=v_info_array.count+1;
113 v_info_array.extend;
114 v_info_array(v_count):=(v_offset);
115 v_count:=v_count+1;
116 v_info_array.extend;
117 v_info_array(v_count):=(1003);
```

```
118 v_count:=v_count+1;
119 v_info_array.extend;
120 v_info_array(v_count):=(1);
121
122 --extend ordinate-array
123 x:=v_Fordinate.count;
124 FOR v_counter IN 1..x LOOP
125 v_ordinate.extend;
126 v_ordinate(v_offset):=v_Fordinate(v_counter);
127 v_offset:=v_offset+1;
128 END LOOP;
129 END IF;
130 END LOOP;
131 END IF;
132
133
134 --get neg. oriented faces of outer shell
135 IF tab_NEGface.count>0 THEN
136 FOR i IN tab_NEGface.FIRST..tab_NEGface.LAST LOOP
137 v_face:=tab_NEGface(i);
138 --dbms_output.put_line('(neg) face: '||v_face);
139
140 SELECT count(*) INTO v_innerrings FROM ring WHERE InOut='I' AND
    face_id_ref=v_face;
141 IF v_innerrings>1 THEN dbms_output.put_line ('more than 1 inner ring in
    face: '||v_face||' (geen solid geconstrueerd)'); VOLUME:=NULL;
142 ELSIF v_innerrings=1 THEN dbms_output.put_line ('1 inner ring, face:
    '||v_face);
143 v_nrfaces:=v_nrfaces+1;
144 --dbms_output.put_line('number of faces '||v_nrfaces);
145 v_info_array(6):=(v_nrfaces);
146 deleteINNERRINGS(v_face,v_face1ordinate,v_face2ordinate);
147
148 --adjust array-info for face1
149 v_offset:=v_ordinate.count+1;
150 v_count:=v_info_array.count+1;
151 v_info_array.extend;
152 v_info_array(v_count):=(v_offset);
153 v_count:=v_count+1;
154 v_info_array.extend;
155 v_info_array(v_count):=(1003);
156 v_count:=v_count+1;
157 v_info_array.extend;
158 v_info_array(v_count):=(1);
159 --extend ordinate-array for face1
160 x:=v_face1ordinate.count;
161 v_loop:=(x/3);
162 FOR v_counter IN 1..v_loop LOOP
163 v_count:=x-2;
164 --dbms_output.put_line ('offset: '||v_offset);
165 --dbms_output.put_line ('count: '||v_count);
166 v_ordinate.extend;
167 v_ordinate(v_offset):=v_face1ordinate(v_count);
168 v_offset:=v_offset+1;
169 v_count:=x-1;
170 --dbms_output.put_line ('offset: '||v_offset);
171 --dbms_output.put_line ('count: '||v_count);
172 v_ordinate.extend;
173 v_ordinate(v_offset):=v_face1ordinate(v_count);
174 v_offset:=v_offset+1;
175 v_count:=x;
176 --dbms_output.put_line ('offset: '||v_offset);
177 --dbms_output.put_line ('count: '||v_count);
```

```
178 v_ordinate.extend;
179 v_ordinate(v_offset):=v_face1ordinate(v_count);
180 v_offset:=v_offset+1;
181 x:=x-3;
182 END LOOP;
183
184 --adjust array-info for face2
185 v_offset:=v_ordinate.count+1;
186 v_count:=v_info_array.count+1;
187 v_info_array.extend;
188 v_info_array(v_count):=(v_offset);
189 v_count:=v_count+1;
190 v_info_array.extend;
191 v_info_array(v_count):=(1003);
192 v_count:=v_count+1;
193 v_info_array.extend;
194 v_info_array(v_count):=(1);
195 --extend ordinate-array for face2
196 x:=v_face1ordinate.count;
197 v_loop:=(x/3);
198 FOR v_counter IN 1..v_loop LOOP
199 v_count:=x-2;
200 --dbms_output.put_line ('offset: '||v_offset);
201 --dbms_output.put_line ('count: '||v_count);
202 v_ordinate.extend;
203 v_ordinate(v_offset):=v_face2ordinate(v_count);
204 v_offset:=v_offset+1;
205 v_count:=x-1;
206 --dbms_output.put_line ('offset: '||v_offset);
207 --dbms_output.put_line ('count: '||v_count);
208 v_ordinate.extend;
209 v_ordinate(v_offset):=v_face2ordinate(v_count);
210 v_offset:=v_offset+1;
211 v_count:=x;
212 --dbms_output.put_line ('offset: '||v_offset);
213 --dbms_output.put_line ('count: '||v_count);
214 v_ordinate.extend;
215 v_ordinate(v_offset):=v_face2ordinate(v_count);
216 v_offset:=v_offset+1;
217 x:=x-3;
218 END LOOP;
219
220
221 ELSE SELECT ringordinates INTO v_Fordinate FROM ringordinates WHERE
    ring_id IN (SELECT ring_id FROM ring WHERE face_id_ref=v_face AND
    InOut='O');
222 --dbms_output.put_line('geen inner rings');
223
224 --adjust array-info
225 v_offset:=v_ordinate.count+1;
226 v_count:=v_info_array.count+1;
227 v_info_array.extend;
228 v_info_array(v_count):=(v_offset);
229 v_count:=v_count+1;
230 v_info_array.extend;
231 v_info_array(v_count):=(1003);
232 v_count:=v_count+1;
233 v_info_array.extend;
234 v_info_array(v_count):=(1);
235 --dbms_output.put_line ('face: '||v_face);
236
237 --extend ordinate-array
238 x:=v_Fordinate.count;
```

```
239 --dbms_output.put_line ('number of ordinates: '||x);
240 v_loop:=(x/3);
241 FOR v_counter IN 1..v_loop LOOP
242 v_count:=x-2;
243 --dbms_output.put_line ('offset: '||v_offset);
244 --dbms_output.put_line ('count: '||v_count);
245 v_ordinate.extend;
246 v_ordinate(v_offset):=v_Fordinate(v_count);
247 v_offset:=v_offset+1;
248 v_count:=x-1;
249 --dbms_output.put_line ('offset: '||v_offset);
250 --dbms_output.put_line ('count: '||v_count);
251 v_ordinate.extend;
252 v_ordinate(v_offset):=v_Fordinate(v_count);
253 v_offset:=v_offset+1;
254 v_count:=x;
255 --dbms_output.put_line ('offset: '||v_offset);
256 --dbms_output.put_line ('count: '||v_count);
257 v_ordinate.extend;
258 v_ordinate(v_offset):=v_Fordinate(v_count);
259 v_offset:=v_offset+1;
260 x:=x-3;
261 END LOOP;
262 END IF;
263 END LOOP;
264 END IF;
265
266
267 --check for inner shells
268 IF tab_Ishell.count>0 THEN
269 FOR i IN tab_Ishell.FIRST..tab_Ishell.LAST LOOP
270 v_Ishell:=tab_Ishell(i);
271 --dbms_output.put_line ('inner shell: '||v_Ishell);
272 --get faces of inner shell
273 SELECT face_id BULK COLLECT INTO tab_POSface FROM FACE WHERE
    shell_id_ref_pos=v_Ishell;
274 SELECT face_id BULK COLLECT INTO tab_NEGface FROM FACE WHERE
    shell_id_ref_neg=v_Ishell;
275 v_nrfaces:=tab_POSface.count+tab_NEGface.count;
276 --dbms_output.put_line ('number of faces: '||v_nrfaces);
277 --dbms_output.put_line ('pos oriented faces: '||tab_POSface.count);
278 --dbms_output.put_line ('neg oriented faces: '||tab_NEGface.count);
279
280
281 --adjust array-info for inner shell
282 v_offset:=v_ordinate.count+1;
283 --dbms_output.put_line ('offset: '||v_offset);
284 v_count:=v_info_array.count+1;
285 --dbms_output.put_line ('count: '||v_count);
286 v_info_array.extend;
287 v_info_array(v_count):=(v_offset);
288 v_count:=v_count+1;
289 v_info_array.extend;
290 v_info_array(v_count):=(2006);
291 v_count:=v_count+1;
292 v_info_array.extend;
293 v_info_array(v_count):=(v_nrfaces);
294
295 --get pos. oriented faces first (of inner shell)
296 IF tab_POSface.count>0 THEN
297 FOR i IN tab_POSface.FIRST..tab_POSface.LAST LOOP
298 v_face:=tab_POSface(i);
299 SELECT ringordinates INTO v_Fordinate FROM ringordinates WHERE ring_id IN
```

```
    (SELECT ring_id FROM ring WHERE face_id_ref=v_face AND InOut='O');
300
301 --adjust array-info
302 v_offset:=v_ordinate.count+1;
303 v_count:=v_info_array.count+1;
304 v_info_array.extend;
305 v_info_array(v_count):=(v_offset);
306 v_count:=v_count+1;
307 v_info_array.extend;
308 v_info_array(v_count):=(2003);
309 v_count:=v_count+1;
310 v_info_array.extend;
311 v_info_array(v_count):=(1);
312
313 --extend ordinate-array
314 x:=v_Fordinate.count;
315 FOR v_counter IN 1..x LOOP
316 v_ordinate.extend;
317 v_ordinate(v_offset):=v_Fordinate(v_counter);
318 v_offset:=v_offset+1;
319 END LOOP;
320 END LOOP;
321 END IF;
322
323 --get neg. oriented faces of inner shell
324 IF tab_NEGface.count>0 THEN
325 FOR i IN tab_NEGface.FIRST..tab_NEGface.LAST LOOP
326 v_face:=tab_NEGface(i);
327 SELECT ringordinates INTO v_Fordinate FROM ringordinates WHERE ring_id IN
    (SELECT ring_id FROM ring WHERE face_id_ref=v_face AND InOut='O');
328
329 --adjust array-info
330 v_offset:=v_ordinate.count+1;
331 v_count:=v_info_array.count+1;
332 v_info_array.extend;
333 v_info_array(v_count):=(v_offset);
334 v_count:=v_count+1;
335 v_info_array.extend;
336 v_info_array(v_count):=(2003);
337 v_count:=v_count+1;
338 v_info_array.extend;
339 v_info_array(v_count):=(1);
340
341 --extend ordinate-array
342 x:=v_Fordinate.count;
343 --dbms_output.put_line ('number of ordinates: '||x);
344 v_loop:=(x/3);
345 FOR v_counter IN 1..v_loop LOOP
346 v_count:=x-2;
347 --dbms_output.put_line ('offset: '||v_offset);
348 --dbms_output.put_line ('count: '||v_count);
349 v_ordinate.extend;
350 v_ordinate(v_offset):=v_Fordinate(v_count);
351 v_offset:=v_offset+1;
352 v_count:=x-1;
353 --dbms_output.put_line ('offset: '||v_offset);
354 --dbms_output.put_line ('count: '||v_count);
355 v_ordinate.extend;
356 v_ordinate(v_offset):=v_Fordinate(v_count);
357 v_offset:=v_offset+1;
358 v_count:=x;
359 --dbms_output.put_line ('offset: '||v_offset);
360 --dbms_output.put_line ('count: '||v_count);
```

```
361 v_ordinate.extend;
362 v_ordinate(v_offset):=v_Fordinate(v_count);
363 v_offset:=v_offset+1;
364 x:=x-3;
365 END LOOP;
366 END LOOP;
367 END IF;
368 END LOOP;
369 END IF;
370
371 --result
372 --dbms_output.put_line ('info-array: '||v_info_array(1));
373 --dbms_output.put_line ('ordinate-array: '||v_ordinate(1));
374 VOLUME:=SDO_GEOMETRY(3008,NULL,NULL,v_info_array,v_ordinate);
375 RETURN VOLUME;
376 RETURN VOLUME;
377 EXCEPTION WHEN NO_DATA_FOUND THEN dbms_output.put_line('no data
    found');VOLUME:=NULL;
378 RETURN VOLUME;
379
380 END;
```

Scripts assistant functions

```
1  //FUNCTION RINGORDINATES//
2
3
4  CREATE OR REPLACE FUNCTION getRINGordinates(i_ring IN NUMBER) RETURN
   SDO_ORDINATE_ARRAY AS
5  TYPE element_type IS TABLE OF NUMBER;
6  tab_node element_type;
7  v_ring NUMBER;
8  v_edge NUMBER;
9  v_node NUMBER;
10 v_orientation VARCHAR2(1);
11 v_startnode NUMBER;
12 v_chkedge NUMBER;
13 v_chknode NUMBER;
14 v_counter NUMBER;
15 v_x NUMBER;
16 v_y NUMBER;
17 v_z NUMBER;
18 v_nredges NUMBER;
19 v_nrordinates NUMBER;
20 v_ordinate SDO_ORDINATE_ARRAY;
21 RING SDO_ORDINATE_ARRAY;
22
23 BEGIN
24
25 --getting started
26 v_ring:=i_ring;
27
28 --set up check: count edges to check with number of ordinates
29 SELECT count(edge_id_ref) INTO v_nredges FROM ring2edge WHERE
   ring_id_ref=v_ring;
30 v_nrordinates:=(v_nredges*3)+3;
31
32 --select a startedge and startnode
33 SELECT min(edge_id_ref) INTO v_edge FROM ring2edge WHERE
   ring_id_ref=v_ring;
34 --dbms_output.put_line('startedge: '||v_edge);
35 SELECT orientation INTO v_orientation FROM ring2edge WHERE
   ring_id_ref=v_ring AND edge_id_ref=v_edge;
36 --dbms_output.put_line('orientation startedge: '||v_orientation);
37 IF v_orientation='-'
38 THEN SELECT endnode, startnode INTO v_startnode, v_node FROM edge WHERE
   edge_id=v_edge;
39 ELSE SELECT startnode, endnode INTO v_startnode, v_node FROM edge WHERE
   edge_id=v_edge;
40 END IF;
41 --dbms_output.put_line('startnode: '||v_startnode);
42
43 --insert startnode in ordinate-array
44 SELECT x INTO v_x FROM node WHERE node_id= v_startnode;
45 SELECT y INTO v_y FROM node WHERE node_id= v_startnode;
46 SELECT z INTO v_z FROM node WHERE node_id= v_startnode;
47 v_ordinate:=sdo_ordinate_array(v_x,v_y,v_z);
48 v_counter:=3;
49
50 --insert the second node in ordinate-array
51 SELECT x INTO v_x FROM node WHERE node_id= v_node;
52 SELECT y INTO v_y FROM node WHERE node_id= v_node;
53 SELECT z INTO v_z FROM node WHERE node_id= v_node;
54 v_counter:=v_counter+1;
55 v_ordinate.extend;
56 v_ordinate(v_counter):=(v_x);
57 v_counter:=v_counter+1;
```

```
58 v_ordinate.extend;
59 v_ordinate(v_counter):=(v_y);
60 v_counter:=v_counter+1;
61 v_ordinate.extend;
62 v_ordinate(v_counter):=(v_z);
63
64 --continue collecting and inserting nodes (collect next node and check
   orientation edge)
65 LOOP
66
67 --search for negative oriented edge
68 BEGIN
69 SELECT startnode,edge_id INTO v_chknode,v_chkedge FROM edge WHERE
   endnode=v_node AND edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
   ring_id_ref=v_ring) AND edge_id <>v_edge;
70 v_node:=v_chknode;
71 v_edge:=v_chkedge;
72 --chk (neg.) orientation
73 SELECT orientation INTO v_orientation FROM ring2edge WHERE
   edge_id_ref=v_chkedge AND ring_id_ref=v_ring;
74 IF v_orientation='+' THEN dbms_output.put_line('1.edge not oriented
   properly: '||v_edge); END IF;
75 EXCEPTION WHEN NO_DATA_FOUND THEN v_chknode:=0;
76 END;
77
78 --search for positive oriented edge (when no negative oriented edge is
   found)
79 IF v_chknode=0 THEN
80 SELECT endnode,edge_id INTO v_chknode,v_chkedge FROM edge WHERE
   startnode=v_node AND edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
   ring_id_ref=v_ring) AND edge_id <>v_edge;
81 v_node:=v_chknode;
82 v_edge:=v_chkedge;
83 --chk (pos.) orientation
84 SELECT orientation INTO v_orientation FROM ring2edge WHERE
   edge_id_ref=v_chkedge AND ring_id_ref=v_ring;
85 IF v_orientation='-' THEN dbms_output.put_line('2.edge not oriented
   properly: '||v_edge); END IF;
86 END IF;
87
88 --insert v_node in ordinate-array
89 SELECT x INTO v_x FROM node WHERE node_id= v_node;
90 SELECT y INTO v_y FROM node WHERE node_id= v_node;
91 SELECT z INTO v_z FROM node WHERE node_id= v_node;
92 v_counter:=v_counter+1;
93 v_ordinate.extend;
94 v_ordinate(v_counter):=(v_x);
95 v_counter:=v_counter+1;
96 v_ordinate.extend;
97 v_ordinate(v_counter):=(v_y);
98 v_counter:=v_counter+1;
99 v_ordinate.extend;
100 v_ordinate(v_counter):=(v_z);
101 EXIT WHEN v_node= v_startnode OR v_ordinate.count=v_nrordinates;
102 END LOOP;
103
104 --check end-startnode
105 IF v_node<>v_startnode THEN dbms_output.put_line('startnode does not
   equal endnode (possibly more than 2-manifold or a wrong orientation of
   the first edge'); END IF;
106 --check number of nodes/ordinates
107 IF v_ordinate.count=v_nrordinates THEN
108 RING:=v_ordinate;
```

```
109 RETURN RING;
110 ELSE RING:=NULL; RETURN RING; END IF;
111 EXCEPTION WHEN NO_DATA_FOUND THEN RING:=NULL;
112 RETURN RING;
113
114 END;
```

```
1  //FUNCTION GETNORMAL//
2
3
4  CREATE OR REPLACE FUNCTION getNORMAL(i_ring IN NUMBER) RETURN
   NORMAL_ARRAY AS
5
6  NORMAL NORMAL_ARRAY;
7  v_ring NUMBER;
8  x1 NUMBER;
9  y1 NUMBER;
10 z1 NUMBER;
11 x2 NUMBER;
12 y2 NUMBER;
13 z2 NUMBER;
14 x3 NUMBER;
15 y3 NUMBER;
16 z3 NUMBER;
17 x4 NUMBER;
18 y4 NUMBER;
19 z4 NUMBER;
20 a NUMBER:=4;
21 b NUMBER:=5;
22 c NUMBER:=6;
23 d NUMBER:=7;
24 e NUMBER:=8;
25 f NUMBER:=9;
26 vec1x NUMBER;
27 vec1y NUMBER;
28 vec1z NUMBER;
29 vec2x NUMBER;
30 vec2y NUMBER;
31 vec2z NUMBER;
32 normalX NUMBER;
33 normalY NUMBER;
34 normalZ NUMBER;
35 v_Nfactor NUMBER;
36 v_ordinate SDO_ORDINATE_ARRAY;
37
38 BEGIN
39
40 --getting started
41 v_ring:=i_ring;
42
43 --select x,y,z values of four nodes (representing 2 edges)
44 SELECT ringordinates INTO v_ordinate FROM ringordinates WHERE
   ring_id=v_ring;
45 x1:= v_ordinate(1);
46 y1:= v_ordinate(2);
47 z1:= v_ordinate(3);
48 x2:= v_ordinate(4);
49 y2:= v_ordinate(5);
50 z2:= v_ordinate(6);
51 x3:= v_ordinate(a);
52 y3:= v_ordinate(b);
53 z3:= v_ordinate(c);
54 x4:= v_ordinate(d);
55 y4:= v_ordinate(e);
56 z4:= v_ordinate(f);
57
58 --calculate the normal of the ring
59 vec1x:=x2-x1;
60 vec1y:=y2-y1;
61 vec1z:=z2-z1;
```

```
62 vec2x:=x4-x3;
63 vec2y:=y4-y3;
64 vec2z:=z4-z3;
65 normalX:= (vec1y * vec2z) - (vec1z * vec2y);
66 normalY:= -((vec2z * vec1x) - (vec2x * vec1z));
67 normalZ:= (vec1x * vec2y) - (vec1y * vec2x);
68
69 --in case the edges are parallel
70 IF (normalX=0 AND normalY=0 AND normalZ=0) THEN
71 --select x,y,z values of two nodes representing another second edge
72 LOOP
73 SELECT ringordinates INTO v_ordinate FROM ringordinates WHERE
   ring_id=v_ring;
74 a:=a+1;
75 b:=b+1;
76 c:=c+1;
77 d:=d+1;
78 e:=e+1;
79 f:=f+1;
80 x3:= v_ordinate(a);
81 y3:= v_ordinate(b);
82 z3:= v_ordinate(c);
83 x4:= v_ordinate(d);
84 y4:= v_ordinate(e);
85 z4:= v_ordinate(f);
86 --calculate the normal of the ring
87 vec1x:=x2-x1;
88 vec1y:=y2-y1;
89 vec1z:=z2-z1;
90 vec2x:=x4-x3;
91 vec2y:=y4-y3;
92 vec2z:=z4-z3;
93 normalX:= (vec1y * vec2z) - (vec1z * vec2y);
94 normalY:= -((vec2z * vec1x) - (vec2x * vec1z));
95 normalZ:= (vec1x * vec2y) - (vec1y * vec2x);
96 CONTINUE WHEN (normalX=0 AND normalY=0 AND normalZ=0);
97 END LOOP;
98 END IF;
99
100 --normalize normal (to delete the influence of the lenght of a normal)
101 v_Nfactor:= sqrt((normalX*normalX)+(normalY*normalY)+(normalZ*normalZ));
102 IF v_Nfactor=0 THEN NORMAL:=normal_array(0,0,0); RETURN NORMAL;
103 ELSE
104 normalX:=normalX/v_Nfactor;
105 normalY:=normalY/v_Nfactor;
106 normalZ:=normalZ/v_Nfactor;
107
108 --result
109 NORMAL:=normal_array(normalX,normalY,normalZ);
110 RETURN NORMAL;
111 END IF;
112
113 END;
```

```
1  //FUNCTION GETSHELL//
2
3
4  CREATE OR REPLACE FUNCTION getSHELL(i_shell IN NUMBER)
5  RETURN SDO_GEOMETRY AS
6
7  TYPE element_type IS TABLE OF NUMBER;
8  tab_POSface element_type;
9  tab_NEGface element_type;
10 v_shell NUMBER;
11 v_InOut VARCHAR(1);
12 v_face NUMBER;
13 v_nrfaces NUMBER;
14 v_offset NUMBER;
15 v_count NUMBER;
16 v_loop NUMBER;
17 x NUMBER;
18 v_innerrings NUMBER;
19 v_ordinate SDO_ORDINATE_ARRAY;
20 v_face1ordinate SDO_ORDINATE_ARRAY;
21 v_face2ordinate SDO_ORDINATE_ARRAY;
22 v_Fordinate SDO_ORDINATE_ARRAY;
23 v_info_array SDO_ELEM_INFO_ARRAY;
24 SHELL SDO_GEOMETRY;
25
26 BEGIN
27
28 --GETTING STARTED
29 v_shell:=i_shell;
30 --dbms_output.put_line('shell: '||v_shell);
31 v_ordinate:=sdo_ordinate_array();
32 v_info_array:=sdo_elem_info_array(1,1007,1);
33 SELECT InOut INTO v_InOut FROM SHELL WHERE shell_id=v_shell;
34 --dbms_output.put_line('InOut: '||v_InOut);
35
36 --ORIGINALLY AN OUTER SHELL
37 IF v_InOut='O' THEN
38 --dbms_output.put_line('outershell');
39 --get faces of shell
40 SELECT face_id BULK COLLECT INTO tab_POSface FROM FACE WHERE
   shell_id_ref_pos=v_shell;
41 SELECT face_id BULK COLLECT INTO tab_NEGface FROM FACE WHERE
   shell_id_ref_neg=v_shell;
42 v_nrfaces:=tab_POSface.count+tab_NEGface.count;
43 --dbms_output.put_line('nr of faces: '||v_nrfaces);
44
45 --adjust array-info for shell
46 v_info_array.extend;
47 v_info_array(4):=1;
48 v_info_array.extend;
49 v_info_array(5):=1006;
50 v_info_array.extend;
51 v_info_array(6):=(v_nrfaces);
52
53 --get pos. oriented faces first
54 IF tab_POSface.count=0 THEN dbms_output.put_line('no pos. oriented faces
   in shell');
55 ELSE FOR i IN tab_POSface.FIRST..tab_POSface.LAST LOOP
56 v_face:=tab_POSface(i);
57 SELECT count(*) INTO v_innerrings FROM ring WHERE InOut='I' AND
   face_id_ref=v_face;
58 --dbms_output.put_line('count inner rings: '||v_innerrings);
59 IF v_innerrings>1 THEN dbms_output.put_line ('more than 1 inner ring in
```

```
    face: '||v_face||' (geen solid geconstrueerd'); SHELL:=NULL;
60  ELSIF v_innerrings=1 THEN dbms_output.put_line ('1 inner ring, face:
    '||v_face);
61  v_nrfaces:=v_nrfaces+1;
62  --dbms_output.put_line ('number of faces: '||v_nrfaces);
63  v_info_array(6):=(v_nrfaces);
64  deleteINNERRINGS(v_face,v_face1ordinate,v_face2ordinate);
65
66  --adjust array-info for face1
67  v_offset:=v_ordinate.count+1;
68  v_count:=v_info_array.count+1;
69  v_info_array.extend;
70  v_info_array(v_count):=(v_offset);
71  v_count:=v_count+1;
72  v_info_array.extend;
73  v_info_array(v_count):=(1003);
74  v_count:=v_count+1;
75  v_info_array.extend;
76  v_info_array(v_count):=(1);
77  --extend ordinate-array for face1
78  x:=v_face1ordinate.count;
79  FOR v_counter IN 1..x LOOP
80  v_ordinate.extend;
81  v_ordinate(v_offset):=v_face1ordinate(v_counter);
82  v_offset:=v_offset+1;
83  END LOOP;
84
85  --adjust array-info for face2
86  v_offset:=v_ordinate.count+1;
87  v_count:=v_info_array.count+1;
88  v_info_array.extend;
89  v_info_array(v_count):=(v_offset);
90  v_count:=v_count+1;
91  v_info_array.extend;
92  v_info_array(v_count):=(1003);
93  v_count:=v_count+1;
94  v_info_array.extend;
95  v_info_array(v_count):=(1);
96  --extend ordinate-array for face2
97  x:=v_face2ordinate.count;
98  FOR v_counter IN 1..x LOOP
99  v_ordinate.extend;
100 v_ordinate(v_offset):=v_face2ordinate(v_counter);
101 v_offset:=v_offset+1;
102 END LOOP;
103
104 ELSE SELECT ring_ordinates INTO v_Fordinate FROM ring WHERE
    face_id_ref=v_face AND InOut='O';
105
106 --adjust array-info
107 v_offset:=v_ordinate.count+1;
108 v_count:=v_info_array.count+1;
109 v_info_array.extend;
110 v_info_array(v_count):=(v_offset);
111 v_count:=v_count+1;
112 v_info_array.extend;
113 v_info_array(v_count):=(1003);
114 v_count:=v_count+1;
115 v_info_array.extend;
116 v_info_array(v_count):=(1);
117
118 --extend ordinate-array
119 x:=v_Fordinate.count;
```

```
120 FOR v_counter IN 1..x LOOP
121 v_ordinate.extend;
122 v_ordinate(v_offset):=v_Fordinate(v_counter);
123 v_offset:=v_offset+1;
124 END LOOP;
125 END IF;
126 END LOOP;
127 END IF;
128
129 --get neg. oriented faces of shell
130 IF tab_NEGface.count<>0 THEN
131 FOR i IN tab_NEGface.FIRST..tab_NEGface.LAST LOOP
132 v_face:=tab_NEGface(i);
133
134 SELECT count(*) INTO v_innerrings FROM ring WHERE InOut='I' AND
    face_id_ref=v_face;
135 IF v_innerrings>1 THEN dbms_output.put_line ('more than 1 inner ring in
    face: '||v_face||' (geen solid geconstrueerd)'); SHELL:=NULL;
136 ELSIF v_innerrings=1 THEN dbms_output.put_line ('1 inner ring, face:
    '||v_face);
137 v_nrfaces:=v_nrfaces+1;
138 --dbms_output.put_line('number of faces '||v_nrfaces);
139 v_info_array(6):=(v_nrfaces);
140 deleteINNERRINGS(v_face,v_face1ordinate,v_face2ordinate);
141
142 --adjust array-info for face1
143 v_offset:=v_ordinate.count+1;
144 v_count:=v_info_array.count+1;
145 v_info_array.extend;
146 v_info_array(v_count):=(v_offset);
147 v_count:=v_count+1;
148 v_info_array.extend;
149 v_info_array(v_count):=(1003);
150 v_count:=v_count+1;
151 v_info_array.extend;
152 v_info_array(v_count):=(1);
153 --extend ordinate-array for face1
154 x:=v_face1ordinate.count;
155 v_loop:=(x/3);
156 FOR v_counter IN 1..v_loop LOOP
157 v_count:=x-2;
158 --dbms_output.put_line ('offset: '||v_offset);
159 --dbms_output.put_line ('count: '||v_count);
160 v_ordinate.extend;
161 v_ordinate(v_offset):=v_face1ordinate(v_count);
162 v_offset:=v_offset+1;
163 v_count:=x-1;
164 --dbms_output.put_line ('offset: '||v_offset);
165 --dbms_output.put_line ('count: '||v_count);
166 v_ordinate.extend;
167 v_ordinate(v_offset):=v_face1ordinate(v_count);
168 v_offset:=v_offset+1;
169 v_count:=x;
170 --dbms_output.put_line ('offset: '||v_offset);
171 --dbms_output.put_line ('count: '||v_count);
172 v_ordinate.extend;
173 v_ordinate(v_offset):=v_face1ordinate(v_count);
174 v_offset:=v_offset+1;
175 x:=x-3;
176 END LOOP;
177
178 --adjust array-info for face2
179 v_offset:=v_ordinate.count+1;
```

```
180 v_count:=v_info_array.count+1;
181 v_info_array.extend;
182 v_info_array(v_count):=(v_offset);
183 v_count:=v_count+1;
184 v_info_array.extend;
185 v_info_array(v_count):=(1003);
186 v_count:=v_count+1;
187 v_info_array.extend;
188 v_info_array(v_count):=(1);
189 --extend ordinate-array for face2
190 x:=v_face1ordinate.count;
191 v_loop:=(x/3);
192 FOR v_counter IN 1..v_loop LOOP
193 v_count:=x-2;
194 --dbms_output.put_line ('offset: '||v_offset);
195 --dbms_output.put_line ('count: '||v_count);
196 v_ordinate.extend;
197 v_ordinate(v_offset):=v_face2ordinate(v_count);
198 v_offset:=v_offset+1;
199 v_count:=x-1;
200 --dbms_output.put_line ('offset: '||v_offset);
201 --dbms_output.put_line ('count: '||v_count);
202 v_ordinate.extend;
203 v_ordinate(v_offset):=v_face2ordinate(v_count);
204 v_offset:=v_offset+1;
205 v_count:=x;
206 --dbms_output.put_line ('offset: '||v_offset);
207 --dbms_output.put_line ('count: '||v_count);
208 v_ordinate.extend;
209 v_ordinate(v_offset):=v_face2ordinate(v_count);
210 v_offset:=v_offset+1;
211 x:=x-3;
212 END LOOP;
213
214
215 ELSE SELECT ring_ordinates INTO v_Fordinate FROM ring WHERE
    face_id_ref=v_face AND InOut='O';
216
217 --adjust array-info
218 v_offset:=v_ordinate.count+1;
219 v_count:=v_info_array.count+1;
220 v_info_array.extend;
221 v_info_array(v_count):=(v_offset);
222 v_count:=v_count+1;
223 v_info_array.extend;
224 v_info_array(v_count):=(1003);
225 v_count:=v_count+1;
226 v_info_array.extend;
227 v_info_array(v_count):=(1);
228
229 --extend ordinate-array
230 x:=v_Fordinate.count;
231 v_loop:=(x/3);
232 FOR v_counter IN 1..v_loop LOOP
233 v_count:=x-2;
234 v_ordinate.extend;
235 v_ordinate(v_offset):=v_Fordinate(v_count);
236 v_offset:=v_offset+1;
237 v_count:=x-1;
238 v_ordinate.extend;
239 v_ordinate(v_offset):=v_Fordinate(v_count);
240 v_offset:=v_offset+1;
241 v_count:=x;
```

```
242 v_ordinate.extend;
243 v_ordinate(v_offset):=v_Fordinate(v_count);
244 v_offset:=v_offset+1;
245 x:=x-3;
246 END LOOP;
247 END IF;
248 END LOOP;
249 END IF;
250
251
252 --ORIGINALLY AN INNER SHELL
253 ELSIF v_InOut='I' THEN
254 --dbms_output.put_line('innershell');
255 --get faces of shell
256 SELECT face_id BULK COLLECT INTO tab_POSface FROM FACE WHERE
    shell_id_ref_pos=v_shell;
257 SELECT face_id BULK COLLECT INTO tab_NEGface FROM FACE WHERE
    shell_id_ref_neg=v_shell;
258 v_nrfaces:=tab_POSface.count+tab_NEGface.count;
259 --dbms_output.put_line(v_nrfaces);
260
261 --adjust array-info for shell
262 v_info_array.extend;
263 v_info_array(4):=1;
264 v_info_array.extend;
265 v_info_array(5):=1006;
266 v_info_array.extend;
267 v_info_array(6):=(v_nrfaces);
268
269 --get pos. oriented faces first (because originally an inner shell, the
    orientation is the other way around)
270 IF tab_NEGface.count=0 THEN dbms_output.put_line('no pos. oriented faces
    in shell');
271 ELSE FOR i IN tab_NEGface.FIRST..tab_NEGface.LAST LOOP
272 v_face:=tab_NEGface(i);
273 SELECT count(*) INTO v_innerrings FROM ring WHERE InOut='I' AND
    face_id_ref=v_face;
274 --dbms_output.put_line('count inner rings: '||v_innerrings);
275 IF v_innerrings>1 THEN dbms_output.put_line ('more than 1 inner ring in
    face: '||v_face||' (geen solid geconstrueerd'); SHELL:=NULL;
276 ELSIF v_innerrings=1 THEN dbms_output.put_line ('1 inner ring, face:
    '||v_face);
277 v_nrfaces:=v_nrfaces+1;
278 --dbms_output.put_line ('number of faces: '||v_nrfaces);
279 v_info_array(6):=(v_nrfaces);
280 deleteINNERRINGS(v_face,v_face1ordinate,v_face2ordinate);
281
282 --adjust array-info for face1
283 v_offset:=v_ordinate.count+1;
284 v_count:=v_info_array.count+1;
285 v_info_array.extend;
286 v_info_array(v_count):=(v_offset);
287 v_count:=v_count+1;
288 v_info_array.extend;
289 v_info_array(v_count):=(1003);
290 v_count:=v_count+1;
291 v_info_array.extend;
292 v_info_array(v_count):=(1);
293 --extend ordinate-array for face1
294 x:=v_face1ordinate.count;
295 FOR v_counter IN 1..x LOOP
296 v_ordinate.extend;
297 v_ordinate(v_offset):=v_face1ordinate(v_counter);
```

```
298 v_offset:=v_offset+1;
299 END LOOP;
300
301 --adjust array-info for face2
302 v_offset:=v_ordinate.count+1;
303 v_count:=v_info_array.count+1;
304 v_info_array.extend;
305 v_info_array(v_count):=(v_offset);
306 v_count:=v_count+1;
307 v_info_array.extend;
308 v_info_array(v_count):=(1003);
309 v_count:=v_count+1;
310 v_info_array.extend;
311 v_info_array(v_count):=(1);
312 --extend ordinate-array for face2
313 x:=v_face2ordinate.count;
314 FOR v_counter IN 1..x LOOP
315 v_ordinate.extend;
316 v_ordinate(v_offset):=v_face2ordinate(v_counter);
317 v_offset:=v_offset+1;
318 END LOOP;
319
320 ELSE SELECT ring_ordinates INTO v_Fordinate FROM ring WHERE
    face_id_ref=v_face AND InOut='O';
321
322 --adjust array-info
323 v_offset:=v_ordinate.count+1;
324 v_count:=v_info_array.count+1;
325 v_info_array.extend;
326 v_info_array(v_count):=(v_offset);
327 v_count:=v_count+1;
328 v_info_array.extend;
329 v_info_array(v_count):=(1003);
330 v_count:=v_count+1;
331 v_info_array.extend;
332 v_info_array(v_count):=(1);
333
334 --extend ordinate-array
335 x:=v_Fordinate.count;
336 FOR v_counter IN 1..x LOOP
337 v_ordinate.extend;
338 v_ordinate(v_offset):=v_Fordinate(v_counter);
339 v_offset:=v_offset+1;
340 END LOOP;
341 END IF;
342 END LOOP;
343 END IF;
344
345 --get neg. oriented faces of shell (because originally an inner shell,
    the orientation is the other way around)
346 IF tab_POSface.count<>0 THEN
347 FOR i IN tab_POSface.FIRST..tab_POSface.LAST LOOP
348 v_face:=tab_POSface(i);
349
350 SELECT count(*) INTO v_innerrings FROM ring WHERE InOut='I' AND
    face_id_ref=v_face;
351 IF v_innerrings>1 THEN dbms_output.put_line ('more than 1 inner ring in
    face: '||v_face||' (geen solid geconstrueerd)'); SHELL:=NULL;
352 ELSIF v_innerrings=1 THEN dbms_output.put_line ('1 inner ring, face:
    '||v_face);
353 v_nrfaces:=v_nrfaces+1;
354 --dbms_output.put_line('number of faces '||v_nrfaces);
355 v_info_array(6):=(v_nrfaces);
```

```
356 deleteINNERRINGS(v_face,v_face1ordinate,v_face2ordinate);
357
358 --adjust array-info for face1
359 v_offset:=v_ordinate.count+1;
360 v_count:=v_info_array.count+1;
361 v_info_array.extend;
362 v_info_array(v_count):=(v_offset);
363 v_count:=v_count+1;
364 v_info_array.extend;
365 v_info_array(v_count):=(1003);
366 v_count:=v_count+1;
367 v_info_array.extend;
368 v_info_array(v_count):=(1);
369 --extend ordinate-array for face1
370 x:=v_face1ordinate.count;
371 v_loop:=(x/3);
372 FOR v_counter IN 1..v_loop LOOP
373 v_count:=x-2;
374 --dbms_output.put_line ('offset: '||v_offset);
375 --dbms_output.put_line ('count: '||v_count);
376 v_ordinate.extend;
377 v_ordinate(v_offset):=v_face1ordinate(v_count);
378 v_offset:=v_offset+1;
379 v_count:=x-1;
380 --dbms_output.put_line ('offset: '||v_offset);
381 --dbms_output.put_line ('count: '||v_count);
382 v_ordinate.extend;
383 v_ordinate(v_offset):=v_face1ordinate(v_count);
384 v_offset:=v_offset+1;
385 v_count:=x;
386 --dbms_output.put_line ('offset: '||v_offset);
387 --dbms_output.put_line ('count: '||v_count);
388 v_ordinate.extend;
389 v_ordinate(v_offset):=v_face1ordinate(v_count);
390 v_offset:=v_offset+1;
391 x:=x-3;
392 END LOOP;
393
394 --adjust array-info for face2
395 v_offset:=v_ordinate.count+1;
396 v_count:=v_info_array.count+1;
397 v_info_array.extend;
398 v_info_array(v_count):=(v_offset);
399 v_count:=v_count+1;
400 v_info_array.extend;
401 v_info_array(v_count):=(1003);
402 v_count:=v_count+1;
403 v_info_array.extend;
404 v_info_array(v_count):=(1);
405 --extend ordinate-array for face2
406 x:=v_face1ordinate.count;
407 v_loop:=(x/3);
408 FOR v_counter IN 1..v_loop LOOP
409 v_count:=x-2;
410 --dbms_output.put_line ('offset: '||v_offset);
411 --dbms_output.put_line ('count: '||v_count);
412 v_ordinate.extend;
413 v_ordinate(v_offset):=v_face2ordinate(v_count);
414 v_offset:=v_offset+1;
415 v_count:=x-1;
416 --dbms_output.put_line ('offset: '||v_offset);
417 --dbms_output.put_line ('count: '||v_count);
418 v_ordinate.extend;
```

```
419 v_ordinate(v_offset):=v_face2ordinate(v_count);
420 v_offset:=v_offset+1;
421 v_count:=x;
422 --dbms_output.put_line ('offset: '||v_offset);
423 --dbms_output.put_line ('count: '||v_count);
424 v_ordinate.extend;
425 v_ordinate(v_offset):=v_face2ordinate(v_count);
426 v_offset:=v_offset+1;
427 x:=x-3;
428 END LOOP;
429
430 ELSE SELECT ring_ordinates INTO v_Fordinate FROM ring WHERE
    face_id_ref=v_face AND InOut='O';
431
432 --adjust array-info
433 v_offset:=v_ordinate.count+1;
434 v_count:=v_info_array.count+1;
435 v_info_array.extend;
436 v_info_array(v_count):=(v_offset);
437 v_count:=v_count+1;
438 v_info_array.extend;
439 v_info_array(v_count):=(1003);
440 v_count:=v_count+1;
441 v_info_array.extend;
442 v_info_array(v_count):=(1);
443
444 --extend ordinate-array
445 x:=v_Fordinate.count;
446 v_loop:=(x/3);
447 FOR v_counter IN 1..v_loop LOOP
448 v_count:=x-2;
449 v_ordinate.extend;
450 v_ordinate(v_offset):=v_Fordinate(v_count);
451 v_offset:=v_offset+1;
452 v_count:=x-1;
453 v_ordinate.extend;
454 v_ordinate(v_offset):=v_Fordinate(v_count);
455 v_offset:=v_offset+1;
456 v_count:=x;
457 v_ordinate.extend;
458 v_ordinate(v_offset):=v_Fordinate(v_count);
459 v_offset:=v_offset+1;
460 x:=x-3;
461 END LOOP;
462 END IF;
463 END LOOP;
464 END IF;
465 END IF;
466
467
468 --RESULT
469 SHELL:=SDO_GEOMETRY(3008,NULL,NULL,v_info_array,v_ordinate);
470 RETURN SHELL;
471 RETURN SHELL;
472 EXCEPTION WHEN NO_DATA_FOUND THEN SHELL:=NULL;
473 RETURN SHELL;
474
475 END;
```

```
1  //FUNCTION DISSOLVEINNERRINGS//
2
3
4  SET SERVEROUTPUT ON
5
6  CREATE OR REPLACE PROCEDURE deleteINNERRINGS(i_face IN NUMBER,o_ring1 OUT
   SDO_ORDINATE_ARRAY, o_ring2 OUT SDO_ORDINATE_ARRAY) AS
7
8  TYPE element_type IS TABLE OF NUMBER;
9  tab_face element_type;
10 tab_node1 element_type;
11 tab_node2 element_type;
12 v_face NUMBER;
13 v_Iring NUMBER;
14 v_Oring NUMBER;
15 v_edge NUMBER;
16 v_node NUMBER;
17 v_test NUMBER;
18 v_orientation VARCHAR(1);
19 v_max1 NUMBER;
20 v_max2 NUMBER;
21 v_max NUMBER;
22 v_min1 NUMBER;
23 v_min2 NUMBER;
24 v_min NUMBER;
25 v_NodeOmax NUMBER;
26 v_NodeOmin NUMBER;
27 v_NodeImax NUMBER;
28 v_NodeImin NUMBER;
29 v_NodeIO NUMBER;
30 v_ordinate SDO_ORDINATE_ARRAY;
31 v_counter NUMBER;
32 vx NUMBER;
33 vy NUMBER;
34 vz NUMBER;
35 v_chkedge NUMBER;
36 v_chknode NUMBER;
37 v_ringid NUMBER;
38 v_shellpos NUMBER;
39 v_shellneg NUMBER;
40 v_faceid NUMBER;
41
42
43 BEGIN
44
45
46
47
48
49 v_face:=i_face;
50 SELECT ring_id INTO v_Oring FROM ring WHERE face_id_ref=v_face AND
   InOUT='O';
51 SELECT ring_id INTO v_Iring FROM ring WHERE face_id_ref=v_face AND
   InOUT='I';
52 --dbms_output.put_line('face: '||v_face);
53 --dbms_output.put_line('Inner ring: '||v_Iring);
54 --dbms_output.put_line('Outer ring: '||v_Oring);
55
56 --check for nodes used in both inner and outer ring
57 SELECT startnode BULK COLLECT INTO tab_node1 FROM edge WHERE (startnode
   IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge FROM
   edge2shell WHERE shell=11)) OR startnode IN (SELECT endnode FROM edge
   WHERE edge_id IN (SELECT edge FROM edge2shell WHERE shell=11))) AND
```

```
    (startnode IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge
    FROM edge2shell WHERE shell=10)) OR startnode IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge FROM edge2shell WHERE shell=10)));
58  SELECT endnode BULK COLLECT INTO tab_node2 FROM edge WHERE (endnode IN
    (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge FROM edge2shell
    WHERE shell=11)) OR endnode IN (SELECT endnode FROM edge WHERE edge_id IN
    (SELECT edge FROM edge2shell WHERE shell=11))) AND (endnode IN (SELECT
    startnode FROM edge WHERE edge_id IN (SELECT edge FROM edge2shell WHERE
    shell=10)) OR endnode IN (SELECT endnode FROM edge WHERE edge_id IN
    (SELECT edge FROM edge2shell WHERE shell=10)));
59  v_test:=(tab_node1.count+tab_node2.count);
60  IF v_test>1 THEN dbms_output.put_line('more than 1 node is used in both
    inner and outer ring');
61  ELSIF v_test=1 THEN dbms_output.put_line('1 node is used in both inner
    and outer ring');
62  IF tab_node1.count=1 THEN v_nodeIO:=tab_node1(1); ELSE
    v_nodeIO:=tab_node2(1); END IF;
63  ELSE v_nodeIO:=-99;
64  END IF;
65
66
67  --select min/max nodes of outer ring, first by X-ordinate
68  SELECT max(x) INTO v_max1 FROM node WHERE node_id IN (SELECT startnode
    FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring));
69  SELECT max(x) INTO v_max2 FROM node WHERE node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring));
70  IF v_max1>=v_max2 THEN v_max:=v_max1; ELSE v_max:=v_max2; END IF;
71  --dbms_output.put_line('max X outer ring: '||v_max);
72  SELECT min(x) INTO v_min1 FROM node WHERE node_id IN (SELECT startnode
    FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring));
73  SELECT min(x) INTO v_min2 FROM node WHERE node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring));
74  IF v_min1<=v_min2 THEN v_min:=v_min1; ELSE v_min:=v_min2; END IF;
75  --dbms_output.put_line('min X outer ring: '||v_min);
76
77  IF v_min=v_max THEN --chk Y
78  SELECT max(y) INTO v_max1 FROM node WHERE node_id IN (SELECT startnode
    FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring));
79  SELECT max(y) INTO v_max2 FROM node WHERE node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring));
80  IF v_max1>=v_max2 THEN v_max:=v_max1; ELSE v_max:=v_max2; END IF;
81  --dbms_output.put_line('max Y outer ring: '||v_max);
82  SELECT min(y) INTO v_min1 FROM node WHERE node_id IN (SELECT startnode
    FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring));
83  SELECT min(y) INTO v_min2 FROM node WHERE node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring));
84  IF v_min1<=v_min2 THEN v_min:=v_min1; ELSE v_min:=v_min2; END IF;
85  --dbms_output.put_line('min Y outer ring: '||v_min);
86
87  IF v_min=v_max THEN --chk Z
88  SELECT max(z) INTO v_max1 FROM node WHERE node_id IN (SELECT startnode
    FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring));
89  SELECT max(z) INTO v_max2 FROM node WHERE node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
```

```
    ring_id_ref=v_Oring));
 90 IF v_max1>=v_max2 THEN v_max:=v_max1; ELSE v_max:=v_max2; END IF;
 91 --dbms_output.put_line('max Z outer ring: '||v_max);
 92 SELECT min(z) INTO v_min1 FROM node WHERE node_id IN (SELECT startnode
    FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring));
 93 SELECT min(z) INTO v_min2 FROM node WHERE node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring));
 94 IF v_min1<=v_min2 THEN v_min:=v_min1; ELSE v_min:=v_min2; END IF;
 95 --dbms_output.put_line('min Z outer ring: '||v_min);
 96 SELECT min(node_id) INTO v_nodeOmax FROM node WHERE z=v_max AND (node_id
    IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM
    ring2edge WHERE ring_id_ref=v_Oring)) OR node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring)));
 97 SELECT min(node_id) INTO v_nodeOmin FROM node WHERE z=v_min AND (node_id
    IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM
    ring2edge WHERE ring_id_ref=v_Oring)) OR node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring)));
 98 --dbms_output.put_line('max node Outer ring: '||v_nodeOmax);
 99 --dbms_output.put_line('min node Outer ring: '||v_nodeOmin);
100 --select min/max nodes of inner ring
101 SELECT max(z) INTO v_max1 FROM node WHERE node_id IN (SELECT startnode
    FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring));
102 SELECT max(z) INTO v_max2 FROM node WHERE node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring));
103 IF v_max1>=v_max2 THEN v_max:=v_max1; ELSE v_max:=v_max2; END IF;
104 --dbms_output.put_line('max Z inner ring: '||v_max);
105 SELECT min(z) INTO v_min1 FROM node WHERE node_id IN (SELECT startnode
    FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring));
106 SELECT min(z) INTO v_min2 FROM node WHERE node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring));
107 IF v_min1<=v_min2 THEN v_min:=v_min1; ELSE v_min:=v_min2; END IF;
108 --dbms_output.put_line('min Z inner ring: '||v_min);
109 SELECT min(node_id) INTO v_nodeImax FROM node WHERE z=v_max AND (node_id
    IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM
    ring2edge WHERE ring_id_ref=v_Iring)) OR node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring)));
110 SELECT min(node_id) INTO v_nodeImin FROM node WHERE z=v_min AND (node_id
    IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM
    ring2edge WHERE ring_id_ref=v_Iring)) OR node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring)));
111 --dbms_output.put_line('max node inner ring: '||v_nodeImax);
112 --dbms_output.put_line('min node inner ring: '||v_nodeImin);
113
114 ELSE --- continue with Y
115 SELECT min(node_id) INTO v_nodeOmax FROM node WHERE y=v_max AND (node_id
    IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM
    ring2edge WHERE ring_id_ref=v_Oring)) OR node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring)));
116 SELECT min(node_id) INTO v_nodeOmin FROM node WHERE y=v_min AND (node_id
    IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM
    ring2edge WHERE ring_id_ref=v_Oring)) OR node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
```

```
    ring_id_ref=v_Oring)));
117 --dbms_output.put_line('max node Outer ring: '||v_nodeOmax);
118 --dbms_output.put_line('min node Outer ring: '||v_nodeOmin);
119 --select min/max nodes of inner ring
120 SELECT max(y) INTO v_max1 FROM node WHERE node_id IN (SELECT startnode
    FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring));
121 SELECT max(y) INTO v_max2 FROM node WHERE node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring));
122 IF v_max1>=v_max2 THEN v_max:=v_max1; ELSE v_max:=v_max2; END IF;
123 --dbms_output.put_line('max Y inner ring: '||v_max);
124 SELECT min(y) INTO v_min1 FROM node WHERE node_id IN (SELECT startnode
    FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring));
125 SELECT min(y) INTO v_min2 FROM node WHERE node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring));
126 IF v_min1<=v_min2 THEN v_min:=v_min1; ELSE v_min:=v_min2; END IF;
127 --dbms_output.put_line('min Y inner ring: '||v_min);
128 SELECT min(node_id) INTO v_nodeImax FROM node WHERE y=v_max AND (node_id
    IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM
    ring2edge WHERE ring_id_ref=v_Iring)) OR node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring)));
129 SELECT min(node_id) INTO v_nodeImin FROM node WHERE y=v_min AND (node_id
    IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM
    ring2edge WHERE ring_id_ref=v_Iring)) OR node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring)));
130 --dbms_output.put_line('max node inner ring: '||v_nodeImax);
131 --dbms_output.put_line('min node inner ring: '||v_nodeImin);
132 END IF;
133
134 ELSE --- continue with X
135 SELECT min(node_id) INTO v_nodeOmax FROM node WHERE x=v_max AND (node_id
    IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM
    ring2edge WHERE ring_id_ref=v_Oring)) OR node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring)));
136 SELECT min(node_id) INTO v_nodeOmin FROM node WHERE x=v_min AND (node_id
    IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM
    ring2edge WHERE ring_id_ref=v_Oring)) OR node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring)));
137 --dbms_output.put_line('max node Outer ring: '||v_nodeOmax);
138 --dbms_output.put_line('min node Outer ring: '||v_nodeOmin);
139 --select min/max nodes of inner ring
140 SELECT max(x) INTO v_max1 FROM node WHERE node_id IN (SELECT startnode
    FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring));
141 SELECT max(x) INTO v_max2 FROM node WHERE node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring));
142 IF v_max1>=v_max2 THEN v_max:=v_max1; ELSE v_max:=v_max2; END IF;
143 --dbms_output.put_line('max x inner ring: '||v_max);
144 SELECT min(x) INTO v_min1 FROM node WHERE node_id IN (SELECT startnode
    FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring));
145 SELECT min(x) INTO v_min2 FROM node WHERE node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring));
146 IF v_min1<=v_min2 THEN v_min:=v_min1; ELSE v_min:=v_min2; END IF;
```

```
147 --dbms_output.put_line('min x inner ring: '||v_min);
148 SELECT min(node_id) INTO v_nodeImax FROM node WHERE x=v_max AND (node_id
    IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM
    ring2edge WHERE ring_id_ref=v_Iring)) OR node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring)));
149 SELECT min(node_id) INTO v_nodeImin FROM node WHERE x=v_min AND (node_id
    IN (SELECT startnode FROM edge WHERE edge_id IN (SELECT edge_id_ref FROM
    ring2edge WHERE ring_id_ref=v_Iring)) OR node_id IN (SELECT endnode FROM
    edge WHERE edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring)));
150 --dbms_output.put_line('max node inner ring: '||v_nodeImax);
151 --dbms_output.put_line('min node inner ring: '||v_nodeImin);
152 END IF;
153
154 IF v_nodeIO<>-99 THEN
155 IF v_nodeIO=v_nodeImax THEN v_nodeImin:=v_nodeIO; v_nodeOmin:=v_nodeIO;
    ELSE v_nodeImax:=v_nodeIO; v_nodeOmax:=v_nodeIO; END IF;
156 END IF;
157
158
159 --RING 1
160 --start at max of Outer ring
161 BEGIN
162 SELECT edge_id_ref INTO v_edge FROM ring2edge WHERE ring_id_ref=v_Oring
    AND edge_id_ref IN (SELECT edge_id FROM edge WHERE startnode=v_nodeOmax)
    AND orientation ='+';
163 v_orientation:='+';
164 SELECT endnode INTO v_node FROM edge WHERE edge_id=v_edge;
165 EXCEPTION WHEN NO_DATA_FOUND THEN v_edge:=0;
166 END;
167 IF v_edge=0 THEN
168 SELECT edge_id_ref INTO v_edge FROM ring2edge WHERE ring_id_ref=v_Oring
    AND edge_id_ref IN (SELECT edge_id FROM edge WHERE endnode=v_nodeOmax)
    AND orientation ='-';
169 v_orientation:='-';
170 SELECT startnode INTO v_node FROM edge WHERE edge_id=v_edge; END IF;
171 --dbms_output.put_line('startnode outer ring: '||v_nodeOmax);
172 --dbms_output.put_line('startedge outer ring: '||v_edge);
173 --dbms_output.put_line('orientation startedge outer ring:
    '||v_orientation);
174 --dbms_output.put_line('node 2 outer ring: '||v_node);
175
176 --insert startnode in ordinate-array
177 SELECT x INTO vx FROM node WHERE node_id= v_nodeOmax;
178 SELECT y INTO vy FROM node WHERE node_id= v_nodeOmax;
179 SELECT z INTO vz FROM node WHERE node_id= v_nodeOmax;
180 v_ordinate:=sdo_ordinate_array(vx,vy,vz);
181 v_counter:=3;
182 --insert the second node in ordinate-array
183 SELECT x INTO vx FROM node WHERE node_id= v_node;
184 SELECT y INTO vy FROM node WHERE node_id= v_node;
185 SELECT z INTO vz FROM node WHERE node_id= v_node;
186 v_counter:=v_counter+1;
187 v_ordinate.extend;
188 v_ordinate(v_counter):=(vx);
189 v_counter:=v_counter+1;
190 v_ordinate.extend;
191 v_ordinate(v_counter):=(vy);
192 v_counter:=v_counter+1;
193 v_ordinate.extend;
194 v_ordinate(v_counter):=(vz);
195
```

```
196 --continue collecting and inserting nodes
197 IF v_node<>v_nodeOmin THEN LOOP
198 --search for negative oriented edge
199 BEGIN
200 SELECT startnode,edge_id INTO v_chknode,v_chkedge FROM edge WHERE
    endnode=v_node AND edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring) AND edge_id <>v_edge;
201 v_node:=v_chknode;
202 v_edge:=v_chkedge;
203 EXCEPTION WHEN NO_DATA_FOUND THEN v_chknode:=0;
204 END;
205 --search for positive oriented edge (when no negative oriented edge is
    found)
206 IF v_chknode=0 THEN
207 SELECT endnode,edge_id INTO v_chknode,v_chkedge FROM edge WHERE
    startnode=v_node AND edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring) AND edge_id <>v_edge;
208 v_node:=v_chknode;
209 v_edge:=v_chkedge;
210 END IF;
211
212 --insert v_node in ordinate-array
213 --dbms_output.put_line('v_node: '||v_node);
214 SELECT x INTO vx FROM node WHERE node_id= v_node;
215 SELECT y INTO vy FROM node WHERE node_id= v_node;
216 SELECT z INTO vz FROM node WHERE node_id= v_node;
217 --dbms_output.put_line('v_counter: '||v_counter);
218 v_counter:=v_counter+1;
219 v_ordinate.extend;
220 v_ordinate(v_counter):=(vx);
221 v_counter:=v_counter+1;
222 v_ordinate.extend;
223 v_ordinate(v_counter):=(vy);
224 v_counter:=v_counter+1;
225 v_ordinate.extend;
226 v_ordinate(v_counter):=(vz);
227 EXIT WHEN v_node=v_nodeOmin;
228 END LOOP;
229 END IF;
230
231 --continue at min of inner ring
232 BEGIN
233 SELECT edge_id_ref INTO v_edge FROM ring2edge WHERE ring_id_ref=v_Iring
    AND edge_id_ref IN (SELECT edge_id FROM edge WHERE startnode=v_nodeImin)
    AND orientation ='+';
234 v_orientation:='+';
235 SELECT endnode INTO v_node FROM edge WHERE edge_id=v_edge;
236 EXCEPTION WHEN NO_DATA_FOUND THEN v_edge:=0;
237 END;
238 IF v_edge=0 THEN
239 SELECT edge_id_ref INTO v_edge FROM ring2edge WHERE ring_id_ref=v_Iring
    AND edge_id_ref IN (SELECT edge_id FROM edge WHERE endnode=v_nodeImin)
    AND orientation ='-';
240 v_orientation:='-';
241 SELECT startnode INTO v_node FROM edge WHERE edge_id=v_edge; END IF;
242 --dbms_output.put_line('startnode inner ring: '||v_nodeImin);
243 --dbms_output.put_line('startedge inner ring: '||v_edge);
244 --dbms_output.put_line('orientation startedge inner ring:
    '||v_orientation);
245 --dbms_output.put_line('node 2 inner ring: '||v_node);
246
247 --insert startnode in ordinate-array
248 SELECT x INTO vx FROM node WHERE node_id= v_nodeImin;
```

```
249 SELECT y INTO vy FROM node WHERE node_id= v_nodeImin;
250 SELECT z INTO vz FROM node WHERE node_id= v_nodeImin;
251 v_counter:=v_counter+1;
252 v_ordinate.extend;
253 v_ordinate(v_counter):=(vx);
254 v_counter:=v_counter+1;
255 v_ordinate.extend;
256 v_ordinate(v_counter):=(vy);
257 v_counter:=v_counter+1;
258 v_ordinate.extend;
259 v_ordinate(v_counter):=(vz);
260 --insert the second node in ordinate-array
261 SELECT x INTO vx FROM node WHERE node_id= v_node;
262 SELECT y INTO vy FROM node WHERE node_id= v_node;
263 SELECT z INTO vz FROM node WHERE node_id= v_node;
264 v_counter:=v_counter+1;
265 v_ordinate.extend;
266 v_ordinate(v_counter):=(vx);
267 v_counter:=v_counter+1;
268 v_ordinate.extend;
269 v_ordinate(v_counter):=(vy);
270 v_counter:=v_counter+1;
271 v_ordinate.extend;
272 v_ordinate(v_counter):=(vz);
273
274 --continue collecting and inserting nodes
275 IF v_node<>v_nodeImax THEN LOOP
276 --search for negative oriented edge
277 BEGIN
278 SELECT startnode,edge_id INTO v_chknode,v_chkedge FROM edge WHERE
    endnode=v_node AND edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring) AND edge_id <>v_edge;
279 v_node:=v_chknode;
280 v_edge:=v_chkedge;
281 EXCEPTION WHEN NO_DATA_FOUND THEN v_chknode:=0;
282 END;
283 --search for positive oriented edge (when no negative oriented edge is
    found)
284 IF v_chknode=0 THEN
285 SELECT endnode,edge_id INTO v_chknode,v_chkedge FROM edge WHERE
    startnode=v_node AND edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring) AND edge_id <>v_edge;
286 v_node:=v_chknode;
287 v_edge:=v_chkedge;
288 END IF;
289
290 --insert v_node in ordinate-array
291 SELECT x INTO vx FROM node WHERE node_id= v_node;
292 SELECT y INTO vy FROM node WHERE node_id= v_node;
293 SELECT z INTO vz FROM node WHERE node_id= v_node;
294 v_counter:=v_counter+1;
295 v_ordinate.extend;
296 v_ordinate(v_counter):=(vx);
297 v_counter:=v_counter+1;
298 v_ordinate.extend;
299 v_ordinate(v_counter):=(vy);
300 v_counter:=v_counter+1;
301 v_ordinate.extend;
302 v_ordinate(v_counter):=(vz);
303 EXIT WHEN v_node=v_nodeImax;
304 END LOOP;
305 END IF;
306
```

```
307 --insert end node (equalling the start node = max node of outer ring) in
    ordinate array
308 SELECT x INTO vx FROM node WHERE node_id= v_nodeOmax;
309 SELECT y INTO vy FROM node WHERE node_id= v_nodeOmax;
310 SELECT z INTO vz FROM node WHERE node_id= v_nodeOmax;
311 v_counter:=v_counter+1;
312 v_ordinate.extend;
313 v_ordinate(v_counter):=(vx);
314 v_counter:=v_counter+1;
315 v_ordinate.extend;
316 v_ordinate(v_counter):=(vy);
317 v_counter:=v_counter+1;
318 v_ordinate.extend;
319 v_ordinate(v_counter):=(vz);
320 /*
321 SELECT max(ring_id) INTO v_ringid FROM ring;
322 v_ringid:=v_ringid+1;
323 SELECT shell_id_ref_pos, shell_id_ref_neg INTO v_shellpos, v_shellneg
    FROM FACE WHERE face_id=v_face;
324 SELECT max(face_id) INTO v_faceid FROM FACE;
325 v_faceid:=v_faceid+1;
326 INSERT INTO FACE(face_id,shell_id_ref_pos,shell_id_ref_neg) VALUES
    (v_faceid,v_shellpos,v_shellneg);
327 INSERT INTO ring (ring_id, face_id_ref,InOut,ring_ordinates) VALUES
    (v_ringid,v_faceid,'O',v_ordinate);
328 */
329 o_ring1:=v_ordinate;
330
331
332
333 --RING 2
334 --start at min of Outer ring
335 BEGIN
336 SELECT edge_id_ref INTO v_edge FROM ring2edge WHERE ring_id_ref=v_Oring
    AND edge_id_ref IN (SELECT edge_id FROM edge WHERE startnode=v_nodeOmin)
    AND orientation ='+';
337 v_orientation:='+';
338 SELECT endnode INTO v_node FROM edge WHERE edge_id=v_edge;
339 EXCEPTION WHEN NO_DATA_FOUND THEN v_edge:=0;
340 END;
341 IF v_edge=0 THEN
342 SELECT edge_id_ref INTO v_edge FROM ring2edge WHERE ring_id_ref=v_Oring
    AND edge_id_ref IN (SELECT edge_id FROM edge WHERE endnode=v_nodeOmin)
    AND orientation ='-';
343 v_orientation:='-';
344 SELECT startnode INTO v_node FROM edge WHERE edge_id=v_edge; END IF;
345 --dbms_output.put_line('startnode outer ring: '||v_nodeOmin);
346 --dbms_output.put_line('startedge outer ring: '||v_edge);
347 --dbms_output.put_line('orientation startedge outer ring:
    '||v_orientation);
348 --dbms_output.put_line('node 2 outer ring: '||v_node);
349
350 --insert startnode in ordinate-array
351 SELECT x INTO vx FROM node WHERE node_id= v_nodeOmin;
352 SELECT y INTO vy FROM node WHERE node_id= v_nodeOmin;
353 SELECT z INTO vz FROM node WHERE node_id= v_nodeOmin;
354 v_ordinate:=sdo_ordinate_array(vx,vy,vz);
355 v_counter:=3;
356 --insert the second node in ordinate-array
357 SELECT x INTO vx FROM node WHERE node_id= v_node;
358 SELECT y INTO vy FROM node WHERE node_id= v_node;
359 SELECT z INTO vz FROM node WHERE node_id= v_node;
360 v_counter:=v_counter+1;
```

```
361 v_ordinate.extend;
362 v_ordinate(v_counter):=(vx);
363 v_counter:=v_counter+1;
364 v_ordinate.extend;
365 v_ordinate(v_counter):=(vy);
366 v_counter:=v_counter+1;
367 v_ordinate.extend;
368 v_ordinate(v_counter):=(vz);
369
370 --continue collecting and inserting nodes
371 IF v_node<>v_nodeOmax THEN LOOP
372 --search for negative oriented edge
373 BEGIN
374 SELECT startnode,edge_id INTO v_chknode,v_chkedge FROM edge WHERE
    endnode=v_node AND edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring) AND edge_id <>v_edge;
375 v_node:=v_chknode;
376 v_edge:=v_chkedge;
377 EXCEPTION WHEN NO_DATA_FOUND THEN v_chknode:=0;
378 END;
379 --search for positive oriented edge (when no negative oriented edge is
    found)
380 IF v_chknode=0 THEN
381 SELECT endnode,edge_id INTO v_chknode,v_chkedge FROM edge WHERE
    startnode=v_node AND edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Oring) AND edge_id <>v_edge;
382 v_node:=v_chknode;
383 v_edge:=v_chkedge;
384 END IF;
385
386 --insert v_node in ordinate-array
387 --dbms_output.put_line('v_node: '||v_node);
388 SELECT x INTO vx FROM node WHERE node_id= v_node;
389 SELECT y INTO vy FROM node WHERE node_id= v_node;
390 SELECT z INTO vz FROM node WHERE node_id= v_node;
391 --dbms_output.put_line('v_counter: '||v_counter);
392 v_counter:=v_counter+1;
393 v_ordinate.extend;
394 v_ordinate(v_counter):=(vx);
395 v_counter:=v_counter+1;
396 v_ordinate.extend;
397 v_ordinate(v_counter):=(vy);
398 v_counter:=v_counter+1;
399 v_ordinate.extend;
400 v_ordinate(v_counter):=(vz);
401 EXIT WHEN v_node=v_nodeOmax;
402 END LOOP;
403 END IF;
404
405
406 --continue at max of inner ring
407 BEGIN
408 SELECT edge_id_ref INTO v_edge FROM ring2edge WHERE ring_id_ref=v_Iring
    AND edge_id_ref IN (SELECT edge_id FROM edge WHERE startnode=v_nodeImax)
    AND orientation ='+';
409 v_orientation:='+';
410 SELECT endnode INTO v_node FROM edge WHERE edge_id=v_edge;
411 EXCEPTION WHEN NO_DATA_FOUND THEN v_edge:=0;
412 END;
413 IF v_edge=0 THEN
414 SELECT edge_id_ref INTO v_edge FROM ring2edge WHERE ring_id_ref=v_Iring
    AND edge_id_ref IN (SELECT edge_id FROM edge WHERE endnode=v_nodeImax)
    AND orientation ='-';
```

```
415 v_orientation:='-';
416 SELECT startnode INTO v_node FROM edge WHERE edge_id=v_edge; END IF;
417 --dbms_output.put_line('startnode inner ring: '||v_nodeImax);
418 --dbms_output.put_line('startedge inner ring: '||v_edge);
419 --dbms_output.put_line('orientation startedge inner ring:
    '||v_orientation);
420 --dbms_output.put_line('node 2 inner ring: '||v_node);
421
422 --insert startnode in ordinate-array
423 SELECT x INTO vx FROM node WHERE node_id= v_nodeImax;
424 SELECT y INTO vy FROM node WHERE node_id= v_nodeImax;
425 SELECT z INTO vz FROM node WHERE node_id= v_nodeImax;
426 v_counter:=v_counter+1;
427 v_ordinate.extend;
428 v_ordinate(v_counter):=(vx);
429 v_counter:=v_counter+1;
430 v_ordinate.extend;
431 v_ordinate(v_counter):=(vy);
432 v_counter:=v_counter+1;
433 v_ordinate.extend;
434 v_ordinate(v_counter):=(vz);
435 --insert the second node in ordinate-array
436 SELECT x INTO vx FROM node WHERE node_id= v_node;
437 SELECT y INTO vy FROM node WHERE node_id= v_node;
438 SELECT z INTO vz FROM node WHERE node_id= v_node;
439 v_counter:=v_counter+1;
440 v_ordinate.extend;
441 v_ordinate(v_counter):=(vx);
442 v_counter:=v_counter+1;
443 v_ordinate.extend;
444 v_ordinate(v_counter):=(vy);
445 v_counter:=v_counter+1;
446 v_ordinate.extend;
447 v_ordinate(v_counter):=(vz);
448
449 --continue collecting and inserting nodes
450 IF v_node<>v_nodeImin THEN LOOP
451 --search for negative oriented edge
452 BEGIN
453 SELECT startnode,edge_id INTO v_chknode,v_chkedge FROM edge WHERE
    endnode=v_node AND edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring) AND edge_id <>v_edge;
454 v_node:=v_chknode;
455 v_edge:=v_chkedge;
456 EXCEPTION WHEN NO_DATA_FOUND THEN v_chknode:=0;
457 END;
458 --search for positive oriented edge (when no negative oriented edge is
    found)
459 IF v_chknode=0 THEN
460 SELECT endnode,edge_id INTO v_chknode,v_chkedge FROM edge WHERE
    startnode=v_node AND edge_id IN (SELECT edge_id_ref FROM ring2edge WHERE
    ring_id_ref=v_Iring) AND edge_id <>v_edge;
461 v_node:=v_chknode;
462 v_edge:=v_chkedge;
463 END IF;
464
465 --insert v_node in ordinate-array
466 SELECT x INTO vx FROM node WHERE node_id= v_node;
467 SELECT y INTO vy FROM node WHERE node_id= v_node;
468 SELECT z INTO vz FROM node WHERE node_id= v_node;
469 v_counter:=v_counter+1;
470 v_ordinate.extend;
471 v_ordinate(v_counter):=(vx);
```

```
472 v_counter:=v_counter+1;
473 v_ordinate.extend;
474 v_ordinate(v_counter):=(vy);
475 v_counter:=v_counter+1;
476 v_ordinate.extend;
477 v_ordinate(v_counter):=(vz);
478 EXIT WHEN v_node=v_nodeImin;
479 END LOOP;
480 END IF;
481
482 --insert end node (equalling the start node = min node of outer ring) in
    ordinate array
483 SELECT x INTO vx FROM node WHERE node_id= v_nodeOmin;
484 SELECT y INTO vy FROM node WHERE node_id= v_nodeOmin;
485 SELECT z INTO vz FROM node WHERE node_id= v_nodeOmin;
486 v_counter:=v_counter+1;
487 v_ordinate.extend;
488 v_ordinate(v_counter):=(vx);
489 v_counter:=v_counter+1;
490 v_ordinate.extend;
491 v_ordinate(v_counter):=(vy);
492 v_counter:=v_counter+1;
493 v_ordinate.extend;
494 v_ordinate(v_counter):=(vz);
495 /*
496 SELECT max(ring_id) INTO v_ringid FROM ring;
497 v_ringid:=v_ringid+1;
498 SELECT shell_id_ref_pos, shell_id_ref_neg INTO v_shellpos, v_shellneg
    FROM FACE WHERE face_id=v_face;
499 SELECT max(face_id) INTO v_faceid FROM FACE;
500 v_faceid:=v_faceid+1;
501 INSERT INTO FACE(face_id,shell_id_ref_pos,shell_id_ref_neg) VALUES
    (v_faceid,v_shellpos,v_shellneg);
502 INSERT INTO ring (ring_id, face_id_ref,InOut,ring_ordinates) VALUES
    (v_ringid,v_faceid,'O',v_ordinate);
503 */
504 o_ring2:=v_ordinate;
505
506
507
508 END;
```