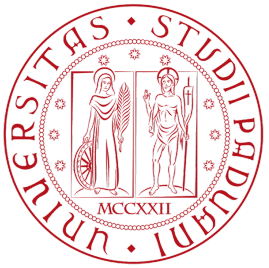


University of Padua

DEPARTMENT OF INFORMATION ENGINEERING
Master Degree Course in Telecommunication Engineering

Data model and software prototype for antenna geometrical information

Thesis advisor:

Professor

Andrea Galtarossa

Candidate:

Rossi Francesca

1057033

Research supervisor:

Senior Antenna Engineer

Marco Sabbadini

Academic year 2014-2015

Acknowledgements

This work is the result of a six months internship (1-07-2013/31-12-2013) at the European Space Research and Technology Center (ESA-ESTEC) in the Antenna and Sub-millimeter Waves Section of the Electrical Engineering Department (TEC-EEA).

Therefore I sincerely want to thank the European Space Agency for the great opportunity and Marco Sabbadini for the limitless support.
I also want to thank Francesca Mioc (MVI) and Poul Erik Frandsen (TICRA) for the concrete and much appreciated help.
Finally my sincere acknowledgement goes to IDS, MVI and TICRA that supported me in this period also with a study grant.

Abstract

The electromagnetic design is a rather long and complex process, that involves several tools each one dealing with its own language and standards. The consequent need of translating and adjusting the data between the tools is not only a time consuming task, but it also sets a limit to the progress in the antenna modeling. To solve this problem the Electromagnetic Data Exchange (EDX) Work Group was founded a decade ago and a common electromagnetic language has been developed. The EDX language is formed by a XML-based Electromagnetic Markup Language (EML), with a simple grammar that is used for the data files, a set of Electromagnetic Data Dictionaries (EDDs), establishing the lexicon, and a software library, the Electromagnetic Data Interface (EDI) for actual data handling.
The implementation of the Structure Data Dictionary (S EDD) is the main theoretical and practical goal of this work. This structured data model includes all the geometrical information and the related physical details, e.g. materials and ports, needed by antenna design tools to perform their job.
To help an organized and complete development of the data set, a Python-based prototype tool, with a minimalistic CAD, was created. This prototype is able to manage the physical structure data model with particular attention to the geometrical and topological information.
The software has been exploited to generate, in an easy and effective way, some complex and complete examples of the Structure Data Dictionary up to a rather detailed model of the Emerald Satellite Geometry Reference.

Contents

List of Figures

List of Tables

List of Abbreviations

BRep : Boundary Representation

CAD : Computer-Aided Design

CSG : Constructive Solid Geometry

DD : Data Dictionary

EDD : Electromagnetic Data Dictionary

EDI : Electromagnetic Data Interface

EDX : Electromagnetic Data Exchange

EML : Electromagnetic Mark-Up Language

Chapter 1

Introduction

This work concern the development of a structured data model for the electromagnetic design process and the implementations of a software prototype tool to test it.

Considering the antenna design, it is quite easy to see how elaborated procedure is: it usually involves many different tools for modelling and simulations and several specialists with a various background. The continuous need of exchanging information and data is overburdened by the need of translating the data model, the conventions and the language used several times taking unnecessary time and resources.
For this reason, over the last decade, the international antenna community began to demand for a common language for electromagnetic purposes, able to smooth and speed up the entire design process.

As a consequence, following the previous attempts promoted by the European Space Agency (ESA), the Electromagnetic Data Exchange (EDX) Working Group had been established to jointly develop the Electromagnetic Data Exchange Language.
The literature on every aspect of the project is copious as a quick search will show. Just to mention some of the most relevant works, the EDX background and requirements are covered in a number of reports and conference papers as [1, 2, 3, 4, 14], the Electromagnetic Data Interface (EDI), which is

a software library allowing standardised access to EDX data file, is detailed in [5, 6, 7, 8, 9]. The Electromagnetic Data Dictionaries (EDDs) already developed are the Fields Data Dictionary, the Current and Meshes Data Dictionary and the Structure one. The first one is detailed in [13] and in [10]. The second one is still under development but some information can be found in [11]. Concerning the last one, only the requirements [15] and some draft documents were available at the beginning of this work, especially [16] and [17].

The rest of the work is organized as follow: Chapter 2 will give a deeper look into the motivations that support this work while Chapter 3 will give a more detailed overview of the background. In Chapter 4 the specific problem of this work is defined. The later chapter, Chapter 5, describes the general theory that lies beneath while Chapter 6 deepen the EDX. The core of the work is presented in Chapters 7 and 8 respectively detailing the structure data model and the software prototype developed. Eventually Chapter 9 demonstrates the results obtained and in Chapter 10 the conclusions are drawn.

Chapter 2

Motivation

As many others engineering products, antenna design is a complex process. Considering antenna for space applications, this is even more true and it is enough to consider the very adverse environment where they work, to frame the problem [12]. The extreme mechanical loads to which antennas are subject during launch, together with the extreme thermal conditions they undergo, make it necessary to take into account both these aspects, including a careful selection of materials and manufacturing process, since the very early stages of the design cycle. Furthermore, the electrical performances of *spaceborn* antenna need to be optimised to satisfy the usually rather tight requirements dictated by the inherent limitations of available power, envelope and mass in spacecrafts.

The main consequence is a design process featuring frequent adjustments to the baseline required by the different discipline involved until convergence to a proven, i.e. ready to fly, solution is reached. The result is a continuous exchange of computer data carrying configuration and performance information among specialty areas. To make the picture even more complicated comes the complexity of the electromagnetic problems underpinning space antenna design. For example, a Ka-band telecommunication antenna, working at 20-30 GHz, often includes a feeding system with sub-millimetric details, a reflector of a few meters and is accommodated on a platform 7-8 metres tall with a 25-30 metres solar-array span. The resulting 10^5 phys-

ical scale range cannot be handled with sufficient accuracy and acceptable computational requirements by any existing electromagnetic modelling tool. Clearly if direct analysis of the whole is impossible the same goes for design optimisation, thus the only possible way is to address the design problem by decomposing it into local sub-problems, each to be addressed with proper means.

As a consequence, electromagnetic design involves many software tools (both for modelling and simulations) which usually use their own different file format and data structures, each one with their own peculiarities, and that usually do not communicate between them.
This situation has clearly a lot of disadvantages such as the time spent converting the data between different format, the unavoidable difference on the models simulated with different softwares, the difficulties in cooperating with people and institutions that come from different backgrounds and, above all, the dissimilarity in viewing the reality.
This last inconvenient originate from the use of different data models, each specialised to a particular physical scale and type of problems, e.g. design of feeder, design of the antenna optics, analysis of antenna interactions. The models include variant treatment of the geometry and of the other physical properties the prototype. For this reason, even when talking only about the geometrical aspect, the data that has to be exchanged between the tools includes not only all the geometrical details but also much other information apparently untied to it, in order to partially overcome the problem.

To solve these problems and to reduce the complexity of the entire process, the main possibility are two: to develop a tool able to perform all the necessary analyses and simulations or to create a uniform data model to allow a seamless exchange of information among tools.
Despite the continuous development of electromagnetic simulations tools, the first alternative seems to be unlikely in short and medium term, whereas the latter one seems promising.

The solution of converging into a common way of defining and describing

the electromagnetic problems may not completely solve, but at least considerably attenuate the difficulties and transform the entire modelling process in a smoother and faster procedure.

At the same time, past experience with the definition of a common model for the description of the electromagnetic fields radiated by antennas has shown that formalising and trying to unify the data models in use offers significant side benefits by establishing more solid basis for the exchange of information also among engineers and to make information produced in the past easier to reuse in new projects.

Chapter 3

Background

Information transfer has been studied in detail for many years and it is not a straightforward issue: even if the quantities to describe are relatively easy to agree on, it is very difficult to find the notational conventions (alphabet, grammar, syntax, etc.) to adopt and to decide how to formally describe the entities in a way that everyone feels comfortable with [14].
The acceptance of a new standard is not obvious at all and it is not a minor issue as the history of the ISO standard for measurements units shows. Moreover in the case of antenna design, notational conventions have to be uniform and able to describe the field distribution or the currents one as well as the geometrical description, i.e. they need to support a rather varied type of data structures originating form the discretization of quite different physical quantities. Fields are typically sampled and quantised on some regular grid of points in space, currents need to be sampled and quantised on meshes laid on conductor surfaces, geometry instead has an inherently hierarchical structural induced by the dimensional scale (point, line, surface, volume) and typically needs to be associated with the hierarchical segmentation of reality used by engineers (and not only) to tame the complexity of reality, in particular the segmentation of systems in sub-systems, equipments, units and so on. Finally, network-like structures are used for RF power distribution in arrays, but also underlay the segmentation of the overall design problem and the re-composition of solutions into a global one. Thus they are

a model for both the partitioning of the electromagnetic problem and for the computational flow to solve it.

To define all the requirements for this purpose, the antenna design universe had been analysed. The most general objects that could be described is an antenna and its environment that can be variegated. The goal usually is to obtain information on the response of the system after having stimulated the physical structure by introducing a signal, a current or a field Figure 3.1.

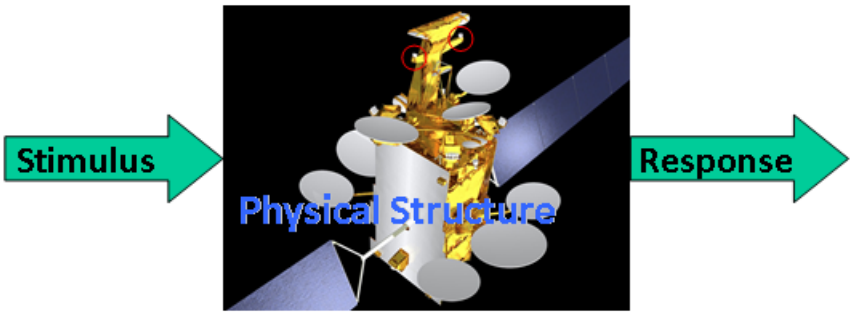


Figure 3.1: Logical Representation of a modelling problem. [14]

It is not by chance that also the response obtained from the system belongs to the same categories of the excitations and this allows the recursive segmentation of the problem, which is also the main cause of the data exchange issue.

It is then clear that the goal of the standardization effort is to describe precisely the stimulus, the physical structure and the response for each desired level of detail. Nevertheless these could be quite complex due to the intrinsic complexity and variety of the electromagnetic phenomena. Luckily, applying the segmentation process common in all system theory, it is possible, at least formally, to consider each part as a black box constituting an *electromagnetic cell* that interacts with the rest of the universe though its excitation and response. In practice this segmentation faces the limitations of available modelling algorithms, which for instance have difficulties in handling cells connected by both electromagnetic fields and current flows. Yet having an effective way to describe the segmentation is a step toward a proper solution of the overall problem.

Thus an Electromagnetic Data Exchange standard can be intended as a language able to robustly delineate the electromagnetic cells and the relative stimuli and responses.

At this point a linguistic problem follows closely behind about how to formally define all the physical and mathematical quantities involved in a robust way. The general requirements identified for the language to reach the goal just outlined can be summed up as follow [14]:

- Readability: human being must easily interpret the descriptions;
- Usability: it must be easy for engineers, software developers and university professors and students to use the descriptions;
- Openness: it is necessary that the language be easily and safely extended to handle future evolutions of the antenna field;
- Completeness: each entry must be fully and univocally described;
- Portability: it must be easy to transfer the information among different computing platform;
- Consistency: it must be possible to check if the description of an entity is complete and free from semantic and syntactic errors (which does not yet means it correctly corresponds to the actual entity).

A deeper analysis starting from the above requirements led to a more detailed set listed in Table 1.

As any other (formal) language, the language in question would need to have an alphabet, a grammar, a syntax and a lexicon. In the Electromagnetic Data Exchange language the Data Dictionary defines the meaning of data and the conventions for their exchange, that is to say that outline exactly and in detail all the elements that shall appear in the data set [19]. Six data sets have been initially identified:

- Fields;

- Induced currents on various geometries;
- Green's functions for layered structures;
- Circuit parameters;
- Modal expansion;
- Geometry.

Only the first two where addressed in the first step and only the first has been actively supported thereafter.

The alphabet and grammar have been chosen among the many widespread one, in particular it was decided to adopt a tagged language so as to ensure future extensions with minimal rework of existing implementations and to base the development on XML [21]. The syntax is instead determined by the underlying information-theoretic structure of the data to be exchanged: ND-arrays, trees and graphs, respectively for sampled fields, hierarchical information and network representations. Also in this case there is quite a large background to draw upon, the syntax adopted is derived from NetCDF [22] and also similar to that adopted later on by HDF5 [23].

Focusing on the geometry there is an important point to underline: the amount of time spent on the geometrical descriptions of antenna has a significant role in the whole electromagnetic modelling process using computational tools. The inevitable fact that geometries have to be changed frequently during design cycles does not make this problem any better. A reasonable improvement would be achieved by using parametric modelling [24]. Unfortunately, the lack of availability of widespread parametric descriptions for CAD applications is not a minor issue. The reasons are quite a few and vary from the difficulties to fit this kind of model in the commonly used algorithms to the lack of a standard approach, from the excessive degree of freedom that a naive approach would induce to the scarcity of rigorous mathematical basis. Finally for the antenna engineering needs the typical high-end professional CAD systems offers far to many features and far too complex parametrisation schemes to present a suitable basis. The most fundamental issue in geometry

parametrisation is the fact that in most cases changing a single parameter in a geometric shape will cause it to change wildly and, eventually to loose integrity. As an example, starting from a square and changing the length of one side it is possible to obtain a trapeze, a triangle, a bow-tie, a polygon collapsed into a segment and, unless a mechanism is in place to avoid it, to break the closed polygon into a three segment straight line plus a single segment one exceeding the total length of the first. Yet in antenna engineering the basic shapes can be parametrised, with just few exceptions, in way that inherently preserves their integrity.

An additional issue, often ignored by CAD systems, as well as by many electromagnetic solvers, is that for a proper solution of Maxwell's equations is usually mandatory to have a topologically sound geometrical description of the structure. Again requirements change from one solution method to another, those based on (quasi-)optical propagation of fields require a coherent orientation of surface normals, while those based on explicit solutions for the induced currents require continuity of segmented surfaces. Finally adjacency relations among geometric shapes are usually needed to build robust algorithms. Unfortunately none of those are explicitly handled by CAD systems, which on the contrary owing to their numerical limitation often produce topologically inconsistent results.

Other methods studied by the branch of mathematics called Computational Geometry can fortunately be adopted to address parametrised geometric descriptions. They are included mainly in two classes of descriptions:

- Constructive Solid Geometry (CSG): based on combining simple (parametric) shapes though boolean operators in order to create more complex geometries.

- Boundary Representation (BRep): based on topological relations between the parts that constitute the shapes.

The question is then which language can be adopted to describe them in a convenient way. During the last forty years, many geometrical data format have been developed and used. Two of the most used standard for the ex-

change of product manufacturing information are IGES and ISO 10303 better knows as STEP (Standard for the Exchange of Product model data).

IGES is a US standard that has been used over twenty years and that had been developed for the exchange of pure geometrical data between computer aided design (CAD) systems. Unfortunately it combines extreme richness with a very basic semantic structure, making it quite inadequate for the handling of complex hierarchical information.

On the other hand STEP has been developed by ISO in over sixteen years with a much broader purpose. In fact, it is intended to cover a wider range of product-related data and cover the entire life-cycle of a product [25]. It use the "neutral file approach" that is to say the data produced by the first tool are translated into the standard language, transmitted as ASCII file and then reconverted by the receiving tool. The entities and their relationship are defined in a information modelling language called EXPRESS [26] that also allow to verify the syntax and to check the possible rules. In principle, it would offer sufficient richness combined with structured semantics for the purpose. However it does not appear to be well suited to handle the large uniform data blocks resulting from fields and currents discretisations, so the question is moved to whether or not it is practically feasible to use (and maintain) two different data exchange language systems within the electromagnetic modelling tools, or it would rather be more effective to use a data model, which is one the one side fully compatible with existing STEP-based standards and on the other one amenable to the Electromagnetic Data Exchange language already used for the exchange of fields and currents data. The second solution implies the capability or reading and writing purely geometric information in STEP format, which is already available in most high-level electromagnetic modelling tools, and the re-use of the existing I/O functions for fields and currents with a different data dictionary and possibly some minor extensions.

As far as we are concerned, in most practical cases the CAD file to be used to solve antenna modelling problems and describing the geometry of a complex structure, e.g. a satellite, is not generated specifically for electromagnetic analysis and includes other, for example mechanical, details that

must be filtered out to make the description suitable for meshing purposes. A further complication is that often these files do not contain material properties and other information, such as topology, needed for the design [28]. As a consequence, a preliminary processing step is virtually always required to adapt CAD data to electromagnetic modelling purpose, thus using STEP, or even IGES, as input and an Electromagnetic Data Exchange language solution in output would not added much complexity to the final product. The pre-processing, called CAD preconditioning, within the EDX group and in their publications, is today rather well understood, defined in some detail and tests have been made to achieve a first validation of the algorithms and procedures. It is to underline that CAD cleaning operations are related to the physics of the problem and they are solver dependent, so baseline requirements can be identified by the knowledge of the specific needs for the different solvers (methods/tools). These baseline requirements are [27]:

- geometry import;
- geometry complexity reduction (Identification of the details to be processed);
- filtering and elimination of parts;
- simplification and transformations;
- management of (multiple) high-level representations of parts;
- generation of local reference systems depending on the EM analysis;
- continuity and consistent orientation of geometrical elements;
- geometry export;
- movable parts;
- antenna excitation ports.

The objective of the data exchange is to handle the information generated by these functions making it available to electromagnetic modelling tools in

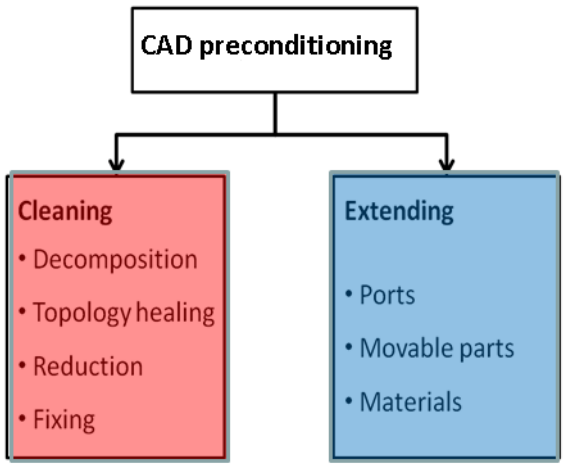


Figure 3.2: CAD preconditioning schema [27].

the form that best suits their needs. In other words, the information content is defined by the pre-processing outcome, while the form (syntax and lexicon) is mostly the object of the formal data modelling effort to be addressed in this work. Clearly such effort finds its main root on the fact that, in most practical cases, the CAD file to be used to solve electromagnetic modelling problems and describing the geometry is not generated specifically for this type of analysis and includes others, e.g. mechanical, details that must be filtered out to make the description suitable for meshing purposes. A further complication is that often these files do not contain material properties and other information, e.g. topology, needed for the same purposes [27].

The challenge has been to achieve a cohesive data model incorporating all these aspects.

General Requirements: -The data format shall reflect as closely as possible the physical structure of reality -The data format shall be based as much as possible on existing and widely accepted data formats, to be selected in agreement with the Agency -The data format shall support incremental definition (refinement and extension) -The data format shall ensure independence, coherence and consistency of different components of the data domain -The data format shall be modular
Implementation Constraints: -The data format shall transitionally accommodate existing data formats to support migration and testing. -The data format shall be as independent as possible from application domain specifics (tool logic and implementation details:) -The data format shall be based on existing mark-up languages (e.g. XML) -All data shall be stored in human-readable form, via text editors or similarly open-domain facilities. Very large data sets may stored in binary form, provided a translator to human-readable form is made available.
Detailed Requirements: -The data format shall allow the definition of meaning of data sets (data and application domain specification) -The data format shall enforce the definition of data sets which are self-contained from the physical meaning point of view -The data format shall accommodate hierarchical structures (nested and recursive) -The data format shall accommodate multidimensional multi-valued data sets -The data format shall accommodate data sets defined on structured and unstructured domains -The data format shall support successive refinements -The data format shall support successive extensions -The data format shall support data sets polymorphism -The data format shall support multiple/mixed formats for individual data set components -The data format shall allow the explicit definition of data blocks structure and format

Table 3.1: List of requirement for an Electromagnetic Data Exchange Standard.

Chapter 4

Problem definition

The increasing use of electromagnetic modelling in antenna design has led to the need for a common way to exchange data between the various software tools available in this field [29].

Following previous attempts promoted by the European Space Agency (ESA), the Electromagnetic Data Exchange (EDX) Working Group was formed for the purpose, composed by a number of institution: the Electromagnetic and Space Division at ESA, the Antenna Centre of Excellence, formed under the 6th EU Framework Programme and the European Antenna Modelling Library team, working under ESA contract.
The outcome is the Electromagnetic Data Exchange (EDX) language. It has been developed following the main lines described in the previous Chapter and it is formed by three main elements: a neutral XML-based Electromagnetic Markup Language (EML), with a simple grammar that is used for the data files, a set of Electromagnetic Data Dictionaries (EDDs) establish the lexicon of the exchange language and a software library, the Electromagnetic Data Interface (EDI), that simplifies the access to data from C++, Fortran and Matlab® programs [29].

Within this project, the main goal of the work is to define the Structure Data Dictionary (S EDD) and to bring it to a sufficient level of maturity to allow its practical demonstration on a set of case studies.

The goal of this Data Dictionary for Structure data exchange is to present a single coherent complete description of a discretised geometry definition, often referred to as "mesh", suited to as many modelling techniques as possible [17]. In order to achieve this goal the data are organized based on:

- Physical (geometrical) as well as topological considerations;
- Ensuring the compatibility with other EDX data dictionaries;
- Covering wide needs in the mesh description;
- Conforming to IEEE standard IEEE Std 145-1993 Antenna Terms[30].

As any other project, three main steps can be individuated:

1. Planning, designing and defining the requirements;
2. Developing the target;
3. Testing and validating the model.

The first step had already been performed and is described in [17] and a complete analysis can be found in [15].

The remaining two stages are described in this work. There were many different possible alternative approaches to proceed, the one selected started from developing a tool using a consolidated EDD (the Field Data Dictionary), followed a trial definition of the new data structure and the iterative check and improvement of the model taking advantage of the tool implemented and using a set of geometries of increasing complexity, up to a full satellite.
Following the validation of the data model, the Structure Data Dictionary had been applied to create some examples.

The second step, development of the data dictionary, is the most complicated. It entails to deeply understand which information are exactly needed to describe any possible scenario without loss of generality that is actually a very difficult goal. To reach it, a continuous debate and comparison with other specialist of the EDX group has been necessary. Moreover, this second step, require to decide how the different information have to be related to

each other to guarantee that no repetition are present while all the necessary references and connections are.
On the other side the development required is based on robust mathematical bases and a well established practical knowledge of the structures to be described. Both need however to be studied. The first, twist around the Computational Geometry and especially the Constructive Solid Geometry and the Boundary Representations. The second needs to consider the tools for antennas designs.

The third and last step, is conceptually simpler and of a much more practical nature: implementing a small tool to handle the EML data files for the new data dictionary and including features to generate or import, process and visualise geometries mimicking the behaviour of real tools for the same purpose. The actual work entails the development of different sets of functions for each purpose and the adjustment and interconnection between them to obtain an organic structured tool.

On the other hand the two steps could not be made in a purely sequential form, the actual work has actually been following an iterative process. It started with the implementation of basic the functionality to handle EDX in the target language selected for the implementation (Python), as a mean to familiarise with the language itself and with its XML libraries as well as to acquire the necessary knowledge about EDX and its workings. Then the data model for the Structure Data Dictionary was addressed, together with the underlying physical and mathematical bases. In this phase also a deeper look into the antenna design process was necessary to gain a more complete understanding of the needs. Next a minimalistic CAD based on the selected BRep representation was implemented, applying previously explored concepts. Finally the two were combined, debugged and verified handling increasingly complex examples. The simpler ones being generated with the minimalistic CAD while larger ones where generated with a commercial CAD (Rhinoceros) and imported using the Polygon File Format or Stanford Triangle Format (.ply) data format, which provides a basic but rather effective way to transfer the faceted geometries used for the testing.

Chapter 5

Theory

5.1 Geometrical Representations

Geometrical modeling is today a rather mature branch of mathematics and computer science and it is often considered part of a discipline called Computational Geometry.
Four major classes of methods are used with success in this field: Parametric Geometry, Constructive Solid Geometry, Boundary Representations and Functional Geometry. The first tackles the complex issue of manipulating parametrically defined objects, as necessary for most engineering applications. The second one deal with the combination of shapes to form more complex ones, while the third one, deal with the problem of providing self-consistent and computationally efficient representations of the geometry of an object. Finally the last one uses continuous real functions of several variables to represent the objects.

5.1.1 Parametric Geometry

The creation and modification of the geometrical descriptions of antenna and platform components, is one of the more time consuming tasks in antenna modeling using computational tools [24]. It is easy too see that geometries have to be changed frequently before reaching the final product design.

The use of parametric descriptions of the geometry, while feasible in prin-

ciple, is limited by the difficulty of translating such description into a numerical model usable by different electromagnetic algorithms. In practice, even if some tools have a way to parametrically define the geometry of objects, there is no standardized approach to deal with this problem. Existing tools use fixed shapes quantified by their "natural" parameters, like a cube and the length of its side, i.e. they rely on implicitly defined relations encoded by the type of shape. This approach is very effective for simple canonical shapes, but becomes increasingly awkward for more complex ones, which in the end can only be defined numerically removing the ability to act parametrically on them.

On the other hand it is particularly hard to find open material on parametric geometry descriptions. There are several reasons for this. Firstly there seems to be no clear way to build a rigorous mathematical formulation of the problem. Second it is quite likely that being this subject strategic for commercial high-end CAD systems many results are kept well protected. Finally the formulations used for CAD system have a level of generality much higher than what would be needed for the antenna-engineering problem. Most of the documentation available deals with the so-called *free-form geometry* parameterization, i.e. with the a-posteriori parameterization of geometrical shapes built, using CSG, by means of free-form primitives, e.g. NURBS. This is far beyond the needs of an antenna modeling application and would require a complete CAD system to be dealt-with [24].

5.1.2 Boundary Representation

The idea behind the Boundary Representation (BRep) is to describe objects though their shells. Usually the shell is a 2-manifold entity with or without boundaries, sometimes constituted by several linked components.
In general these components are described separately and each part is oriented. The only constraint on the orientation is that the possible internal and the external shells of the same component have to have opposite orientation. If boundaries are present they must be oriented accordingly, to preserve consistency of matematical operators, e.g. external vector products.

The oldest data structure for BRep is the Baumfart's *winged-edge* data structure [40]. It is mostly based on the interconnections and adjacency relations between edges (line segments) and faces (polygons surrounded by edges). To efficiently describe it let us consider the polyhedron in Figure 5.1 .

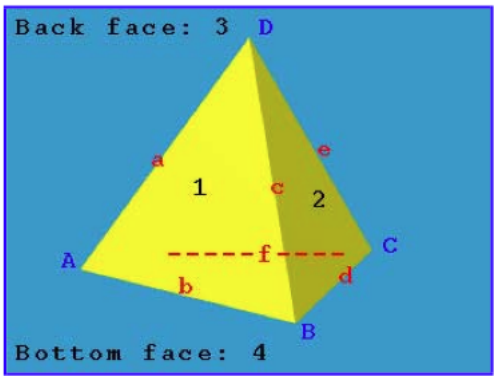


Figure 5.1: Example of polyhedron for BRep description.

The above figure shows the vertices (upper cases), edges (lower cases) and faces (digits) of the structure. Let us consider the edge $a = XY$. This edge has two incident vertices X and Y, and two incident faces 1 and 2. For example, face 1 has its edges a, c and b, and face 2 has its edges a, e and d. Is is to note that the order is clockwise viewed from outside of the solid. If the direction of the edge is from X to Y, faces 1 and 2 are on the right and left side of edge a, respectively. To capture the ordering of edges correctly, additional information are needed. Since edge a is traversed once when traversing face 1 and traversed a second time when traversing face 2, it is used twice in different directions. For example, when traversing the edges of face 1, the predecessor and successor of edge a are edge b and edge c, and when traversing the edges of face 2, the predecessor and successor of edge a are edge d and edge e. Note that although there are four edges incident to vertex X, only three of them are used when finding faces incident to edge a. Therefore, for each edge, the following information are important:

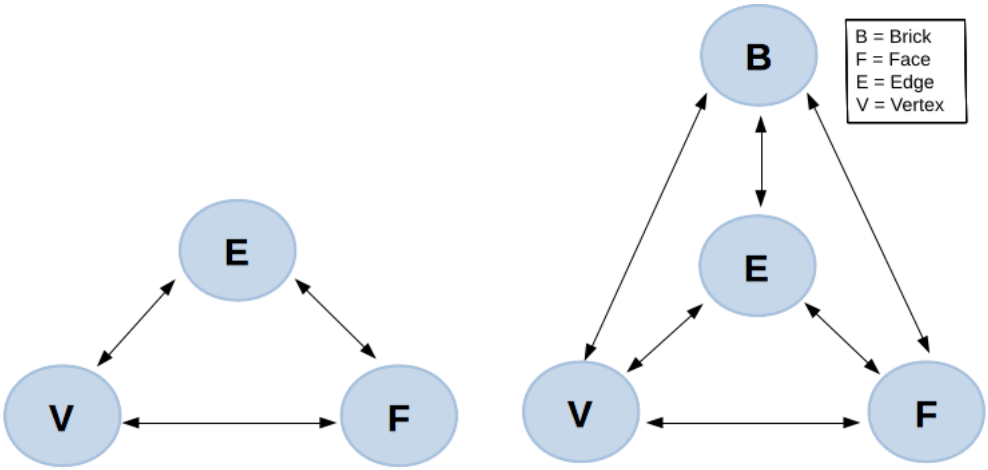
- its vertices;
- its left and right faces;

- the predecessor and successor when traversing its left face;
- the predecessor and successor when traversing its right face.

In the case of objects with holes, inner loops are solved imposing the outer boundary with a clockwise order, while its inner loops, will be ordered counter clockwise.
All these data, and the one regarding the vertices and the faces, have to be collected in tables, usually implemented as matrices or linked lists.

If we consider only the data about edges and vertices, we obtain the 2D BRep that can describe only bidimensional object. To move to tridimensional entities it is enough to add the information about the faces, obtaining the 3D Brep shown in Figure 5.2 (a). By analogy, it is possible to extend the data structure to segmented volumes, which can be considered as the shell (boundary) of a 4D iper-volume, using the 4D Brep structure (Figure 5.2 (b)) that is obtained from the previous one simply adding the brick node and the corresponding relations. This last representation allows full navigation of the iper-surface bounding the shape. Segmented volumetric descriptions are typically necessary in practice to handle antennas and other objects including dielectric parts, which are penetrated by the electromagnetic field and need to be modelled accordingly.
In Figure 5.2 each table is represented as a circle and the complete structure is a dense graph. Although these relations are highly redundant, only a subset of them are actually required to fully describe the topology and the geometry of an object.

A slightly modified version of the direct 4DBrep extension of the Winged-Edge structure appears to be more efficient and is the one actually used for this work (Figure 5.3). In this structure each edge is split in two logical components (called *siblings*), each one associated to one of the two incident faces (called *sheets*). First, the two halves can be oriented in such a way to form counter clockwise cycles around each face, something very useful in practical applications. Second and more relevant in our case, when moving to a higher number of dimensions, i.e. introducing adjacent volumes (3-cells), it is possible to increase the multiplicity of the siblings without altering the



(a) (b)

Figure 5.2: BRep schemas: (a) 3D BRep, (b) 4D BRep.

basic structure of the 3-cell. In this way multiplicity of the basic relations are not changed and the same well-known and mathematically sound algorithms can be used within each 3-cell. Possibly, a number of *nodes* are associated to each vertex, each node being placed at the end of a set of incident siblings within a same 3-cell.

The main advantage of BReps is that it is possible, although not necessarily straightforward, to define algorithms performing all types of geometrical manipulations on them. For example, the basic Boolean operators, union, intersection and different, involve the identification of intersections and the generation of the associated additional faces, edges and vertices resulting in a new BRep for the combined shape. Other transformations may or may not require a modification of the topologic description. For instance, a change in the shape size or a linear deformation (shear transform) do not alter the topology and therefore do not require changes of the BRep, except for the vertex coordinates. Typical CAD manipulations do not involve non-linear geometrical transformations and result in relatively simple manipulation algorithms, making Breps very useful in this domain.

Finally in is important to note that BRep offer a nutural way to handle the surface and volume discretisation used in most electromagnetic mod-

elling algorithms, usually referred to as meshes. These are typically based on faceted approximations, mostly with triangular or quadrangular facets, for 2D and on pyramidal or brick-shaped elements for 3D.

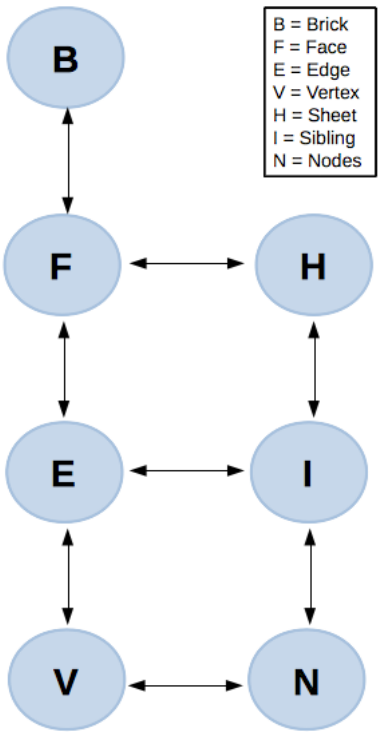


Figure 5.3: Extended 4D Brep.

To conclude, it of interest to highlight as the constraint on the boundaries can be relax to describe non-manifold objects. This entities have the property of not being able to distinguish, at every point on the boundary, a small enough sphere divided into two pieces, one inside and one outside the object. An important sub-class of non-manifold models are sheet objects which are used to represent thin-plate objects quite common in antenna modeling.

5.1.3 Function Representation

Function Representation (FRep) was introduced in [32] as a uniform representation of multidimensional geometric objects. An object, meaning a point set in multidimensional space, is defined by a single continuous real-valued

function of point coordinates F which is evaluated at the given point by a procedure traversing a tree structure with primitives in the leaves and operations in the nodes of the tree. The points where the function have positive values belong to the object, otherwise they are outside of the shape.

The geometric domain of FRep in tridimensional space includes solids with non-manifold models and lower-dimensional entities (surfaces, curves, points) defined by zero value of the function. A primitive can be defined by an equation or by a "black box" procedure [33] converting point coordinates into the function value. Solids bounded by algebraic surfaces, skeleton-based implicit surfaces, and convolution surfaces, as well as procedural objects (such as solid noise), and voxel objects can be used as primitives (leaves of the construction tree). In the case of a voxel object (discrete field), it should be converted to a continuous real function, for example, by applying the trilinear or higher-order interpolation. Many operations such as set-theoretic, blending, offsetting, projection, non-linear deformations, metamorphosis, sweeping, hypertexturing, and others, have been formulated for this representation in such a manner that they yield continuous real-valued functions as output, thus guaranteeing the closure property of the representation. R-functions provide C^k continuity for the functions exactly defining the set-theoretic operations. Because of this property, the result of any supported operation can be treated as the input for a subsequent operation; thus very complex models can be created in this way from a single functional expression. FRep modeling is supported by the special-purpose language HyperFun [34].

Although quite powerful FReps are not very suited for CAD purposes and a combination of BReps and CSG are typically preferred.

5.1.4 Constructive Solid Geometry

The Constructive Solid Geometry (CSG) is probably the most intuitive way of geometrically describing objects and it is therefore the preferred approach for modern CAD tools, at least in the user interface. It is a solid modeling method that combines simple solid shapes to build more complex ones using Booleans operations like union, intersection and difference. The most com-

mon data structure used to store these data, is a binary tree where leaves are the solid shapes, correctly sized and positioned and each internal node is an operator that combine its two leaves.
Algorithm to transform this representation into low-level geometric primitives, typically of the BRep type, complete the data structure.

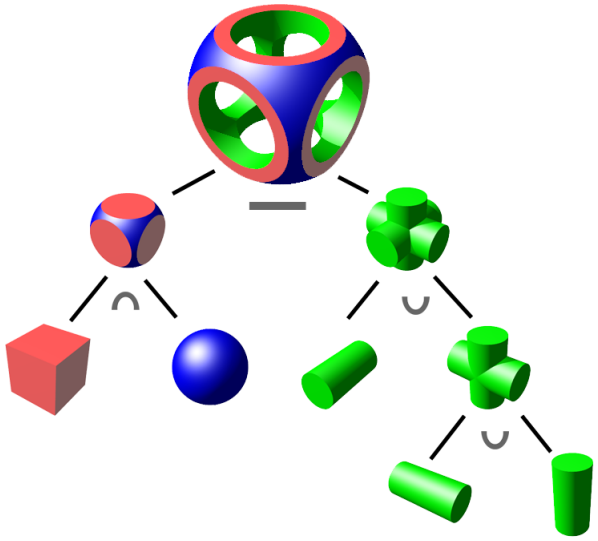


Figure 5.4: Example of CSG
Image from en.wikipedia.org.

5.2 Mark-Up Languages

Many robust data exchange solutions satisfy the general language requirements listed above (Table 3.1), but past experiences has shown that not even the starting point is trivial. On one side a simple format based on a line-by-line description would have been limiting, on the other side high-performances solutions (like NetCDF), which had indeed been experimented in previous projects and proven to be quite effective, would have been difficult to use for non-expert users, in particular for University students and researchers, who are key in the continuing development of the leading-edge electromagnetic modelling algorithms needed in the antenna field. After all the considerations the solution chosen is an XML-based format that satisfy almost all the previous basic requirement and enjoys a well-established and very general framework. Actually the only severe, but not impossible to overcome, limitation of XML is in the handling of large data sets, for which its text-based format is largely unsuitable. Yet various options exist for the parallel use of a binary file format and the use of HDF5 appears to be one of the most likely candidates.

The Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). It was originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data. XML defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. Its strong points are the simplicity, generality, and usability of the language.

For this reason since 2009, hundreds of document formats using XML syntax have been developed, including RSS, Atom, SOAP, and XHTML. XML-based formats have become the default for many office-productivity tools, including Microsoft Office (Office Open XML), OpenOffice.org and LibreOffice (OpenDocument), and Apple's iWork. XML has also been employed as the base language for communication protocols, such as XMPP.

5.3 Python Programming Language

Python is a powerful dynamic programming language that is used in a wide variety of application domains. This language was adopted mainly because it offered in a single package all the main features required for the development such as: a very clear and readable syntax, an intuitive object orientation, full modularity, supporting hierarchical packages, very high-level dynamic data types and extensions and modules easily written in C, C++ [35].

Although not strictly required here, one of the most important properties of this language, is that Python is available for all major operating systems (Windows, Linux/Unix, OS/2, Mac, Amiga and Java virtual machine) and the same source code run unchanged across all implementations.

Furthermore owing to its flexibility it is also commonly adopted as scripting language in CAD systems, including those embedded in some leading-edge electromagnetic modelling tools.

Finally, it offers a rather smooth learning curve, making it easy to build relatively complex applications without a large programming experience.

5.3.1 Python tool-kits

An additional and valued feature of the Python language is to have several thousands of additional modules and companion tool-kits that cover almost every desirable aspect.
For what concern this work, especially four modules have been used since their very attractive properties for realizing a fast prototyping tool.

- PLY: it is an implementation of a lexical analysers and YACC (Yet Another Compiler Compiler)) for Python. It is entirely implemented in Python and uses LR-parsing which is reasonably efficient and well suited for larger grammars [36].

- LXML: it is a library to process XML and HTML in Python. It binds for the C libraries libxml2 and libxslt and it combines the speed and XML feature completeness of these libraries with the simplicity of a native Python API [37].

- Matplotlib: it is a Python 2D plotting library which produces quality figures in a variety of hardcopy formats and interactive environments across platforms. This tool-kit can generate many different kind of plots (histograms, power spectra, bar charts, errorcharts, scatterplots, etc) in a very compact way [38].

- NumPy: it allow scientific computing with Python. Its major features are a powerful N-dimensional array object, tools for integrating C/C++ and Fortran code, useful linear algebra, Fourier transform and random number capabilities. NumPy can also be used as an efficient multi-dimensional container of generic data and allows also to define arbitrary data-types [39].

5.4 Antenna design process

Designing an antenna is a long and complex process that requires several steps backwards and forwards using different tools until the desired result is reached. The information transferred between the tools is usually embedded in a single logical unit: the *model*.
The model should combine the property and behavior data as well as their implicit links; it is to say that it should describe the physical structure, i.e. the antenna, from all relevant point of view. Unfrotunately this appears to be seldom the case today, e.g. most commercial electromagnetic modelling tools are far from such completeness and usability level, as they only cater for basic descriptions of the antenna structure, either on a purely numeric basis (discretised geometry) or using high-level implicit definitons, and of the resulting fields and currents, typically in purely numeric form.

From the deep and complete analysis reported in [42] the basic structure of a design environment can be penciled as in Figure 5.5. Using a bottom to top approach, it can be found a *Data Base*, that has to be considered distributed, introducing the data standard representation which is necessary for their transfer. The second layer is the *Network Layer* supporting the information transfer as main task. This last is followed by the *Application Management*

layer organizing, among the rest, the simulating modeling and the utilities. After the *User Interface*, which is simply a GUI with its own standard, the *Application Software* is set. This level is constituted by the application level packages and their integrability is guaranteed by the implementation of a standard strictly linked to the previous layers. The top level, eventually, is the *User's Tool* that collect all the instrument and preferences that each user develop as support to his work.

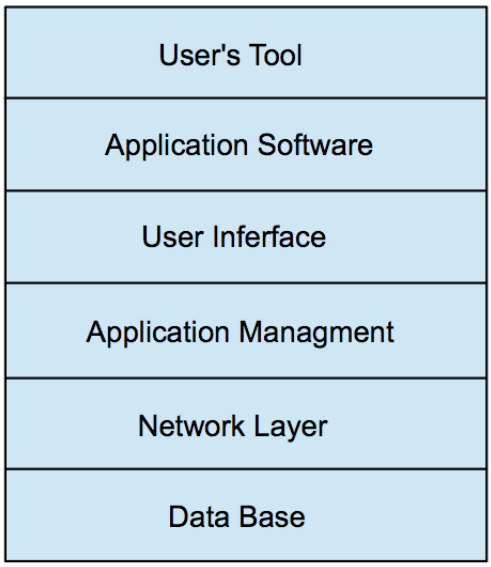


Figure 5.5: Basic structure of an integrated design environment.

In this scenario data transfer is clearly crucial for the efficiency of the the process. Moreover a quick access to all information, both data and control, is fundamental as in any other engineering application.
When designing an antenna at least three different engineering fields are involved: the electromagnetic, the mechanical and the measurement and test area. Each of those needs specific input (format and content) and provide specific output that will constitute the input data for the following step of the process. Therefore the type and the quantity of information used in the different project areas change a lot and the most appealing solution is possibly to create a "container" with all the data and the instruments to easily extract all and only the data corresponding at each specific step.

Today each antenna engineer has to build the container and often needs to manipulate the data to pass information form one tool to another. Thus a first step is to actully define a way to make sure all tools can directly use the data. The long-term objective is to reach a high-level of interoperability among the different tools in such a way to allow the user to build custom design procedures by assembling the different tools in a circuit-like fashion, as illustrated in Figure 5.6. In such perspective the container becomes the patway among tools, i.e. much more than a unified static storage.

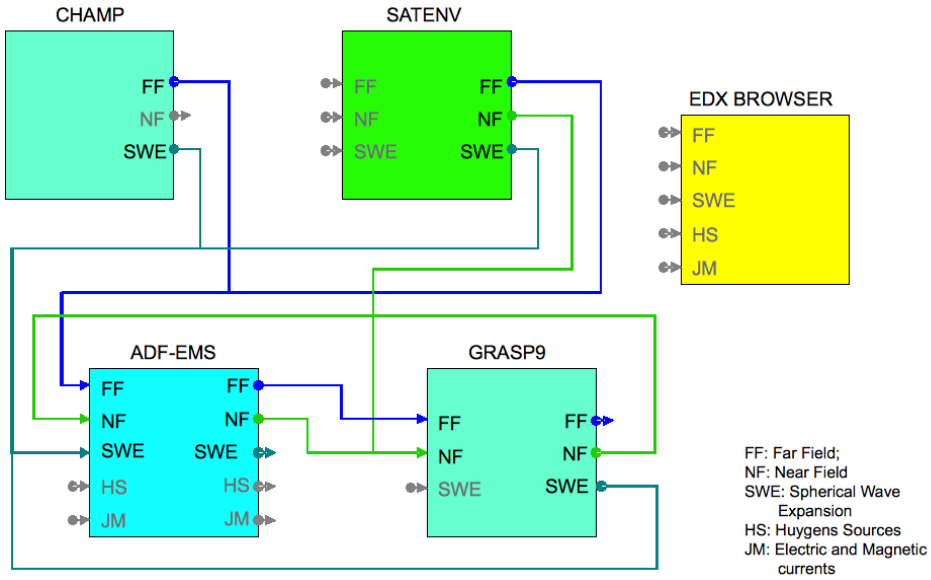


Figure 5.6: Interoperability among the different tool: circuit example.

Chapter 6

Electromagnetic Data Exchange

As already discussed, the Electromagnetic Data Exchange (EDX) is the main outcome of the EAML team work and aims to become a reference for data exchange among electromagnetic modeling software tools, at least in Europe.

In the following sections the components of the language (the Mark-up Language, the Data Dictionaries and the Data Interface) will be described and then a deeper look into the Fields Data Dictionary, developed in the last years, will be taken and eventually, the Structure Data Dictionary will be briefly introduced as will be further discussed in the following chapter as main goal of this work.

6.1 EDX components

Most of the time the raw data to be exchanged are just series of number having no real meaning unless they go together with some more information. The latter often constitutes the most important parts of a data file for the human reader and it is, unfortunately, separated from the data and often implicit, i.e. only available through the User Manual or else. For this reason EDX not only provides the raw data, it most prominently provides their meaning as well as their structure. In other words, a data file includes a complete description of the numerical data it contains.
A set of Electromagnetic Data Dictionaries (EDDs) have been established to

specify the latter information, which conveys the meaning of data, thereby leaving only the values open to host the variable part of the content, i.e. the actual object of the data exchange. The purpose of these dictionaries is to list the information needed to completely described the physical structure, the stimulus and the response, i.e. the input and output data for each computation, at each level of the design procedure. Such approach may appear as unnecessarily complicated as it adds a significant amount of overhead to raw data that may initially be rather simple, e.g. a matrix of complex numbers. However it is often the case that to actually use the data at later step of the design process the overhead becomes precious as without its information content the matrix of numbers is just useless. How to know which was the frequency of the spectral-domain representation of the electric field which samples are recorded in the raw data together with their position in space, if it is not reported together with them? Which was the total radiated power the field carried, i.e. to which power level are the field strength values normalised to? In most cases the engineer needs to keep track of all the information seprately, maybe with very careful naming of the data files, and the main objective of EDX is to overcome this rather primitive behavior.

A software library, the Electromagnetic Data Interface (EDI), allows standardized access to data in the EDX language. A software library relives developers from the burden of writing their own access functions, avoid mistakes and provides a common baseline to which any other implementation can be compared for compliance with the EDX reference. In fact, it can be regarded as the actual embodiment of the language itself.

The last component of the EDX language is the Electromagnetic Markup Language (EML). It specifies how the information is conveyed, e.g. how different quantities are to be named and structured while it does not specify the content, i.e. names, structures and values (symbolic or numerical). It is based on the XML language and its basic blocks are *tags, attributes* and *elements*. The EML segments a data set into four main sections (e.g. in a file):

- Header: contains all the information about its origin and the standard to which it complies.

- Declaration: it specifies the meaning and structure of data, with a collections of *Variables* contained in one or more *Folders*. Each variable belong to a class specified in the corresponding Data Dictionary and host all the components and, if the values are a few, also the raw data to which the variable is referred to.

- Data: hosts all the raw data of the variables defined in the Declaration Section having more than a few numeric values.

- Application Data: this section is intended for tool private data i.e. data that are not part of the data set in question and a tool needs in combination with them.

Finally a set of utilities, the so-called EDX Companion Tools, has been developed and include three tools: a visualizer, a validator and a browser. They provide the additional basic functionality required for the daily use of EDX as well as for its further development.

Nowadays the Electromagnetic Data Exchange language system is fairly well consolidated. Its structure can be summarized with the following formula:

$$\text{EDX} = \text{EDDs} + \text{EML} + \text{EDI}$$

The effort has been made in developing the data exchange model to ensure it would be robust up to the well known limit posed by Godel's theorem on the completeness of formal systems, which states that is impossible to guarantee the syntactic completeness of any formal language, unless you accept contradiction or semantic incompleteness. Actually, this is the main theoretical reason for the segmentation adopted: the trade-off between incompleteness and contraditction is much easier to manage independently for each segment than for the whole. The practical one being, much more prosaically, the need for a segmented approach to tame the extreme complexity of the overall problem. Splitting the language in layers, each having a further internal segmentation, forces the definition of clear interfaces and the early identification of loopholes and omissions, thus making it possible to obtain a more flexible and complete data model (within Godel's constraints).

The first direct consequence of these considerations is that, besides the above mentioned general requirements for a language, more specific technical requirements have to be considered and followed. The new data model shall:

- handle the most common data sets in the field of Antenna and Electromagnetic Engineering;
- have a software library with complete interface for accessing data;
- have easy to understand and human-readable, data files options;
- be flexible, especially with respect to accessing data;
- be able to use multiple representations of the same physical or mathematical quantity;
- have the possibility to store meta-data as well as special data required by some tools;
- be open to future revisions and extensions;
- allow to implement an open and freely available library to access data;
- be able to handle large amounts of data with high performances;
- be independent from the platform used (both software and hardware).

6.2 Fields Data Dictionary

6.2.1 Dictionary Overview

The data dictionary is a collection of definitions specifying a way to convey complete and self-standing information about electromagnetic fields [13]. In this specific case the information covered are in the first place true electromagnetic fields in free-space, i.e. the physical quantities that describe the energy distribution generated by electric charges across a region of free space. They are usually described in engineering text books using their two components **E** and **H**, which have the dimensions of [V/m] and [A/m] respectively. Moreover, other quantities related to fields and used in antenna engineering for the description of antenna radiation are handled (radiation pattern, radiation intensity, directivity, gain, etc).

As can be clearly seen in Figure 6.1, the Field Data Dictionary defines three root quantities (*classes*):

- Near field;
- Far field;
- Spherical Wave Expansion (SWE).

The root quantities are used to simplify data management and act as container for all data relevant to the description of fields and related quantities, like gain and directivity.
Each root class contains three attributes (*SpaceTypeAxis, SpaceTypeAxis, Time Dependency*) that define the convention used in that data set and some other classes also called subclasses. Each of these subclasses are fully defined by other attributes or *components* as define in [13]. Also the mesaurment units, the type and the range for each component of each class and subclass are specified in the Data Dictionary (DD) definition. These values are the default ones and can be *overwritten* simply specifying these quantities and the corresponding new values which are valid only on that data set where they are defined.

The definition of quantities used for spherical wave expansions is not very different from the one valid for near and far field. The major changes are in the substitution of the spatial coordinates with a set of indices, identifying each spherical harmonic, and the replacement of field samples by modal coefficients defined for a very specific decomposition of the field (evanescent and travelling waves of TE and TM type) [13]. Strictly speaking, the latter difference would not affect the structure of the field quantity that would remain a multi-dimensional quantity defined over a certain domain, i.e. the SWE representation could be handled in parallel with the spatial sampling. However, to underline the differences and avoid possible confusions the Field data dictionary defines a separate class. This can be seen as an example of the attempt to overcome Godel's limit by segmentation. This part of the dictionary is based on the definition of Spherical Wave Expansion of Hansen [41].

6.2.2 Example of Fields DD

To better clarify how the Data Dictionary is actually used and how the corresponding EML data file look like, let us consider the example detailed in Appendix A.

The first line specify the XML version and the encoding used:

```
1 <?xml version="1.0" encoding="UTF-8"?>
```

Then, after few lines not particularly interesting to the human reader, the Header Section is defined:

```
6 <Header>
7 <Stamps>
8   <Version>EDI Version 1.00.00</Version>
9   <Format>XML</Format>
10      <DateTime>2006-12-01T12:40:28Z</DateTime>
11   </Stamps>
12   <Origin>
13            <Tool><Name></Name><Version></Version></Tool>
14    <Project></Project>
```

```
15       <User>
16         <Name></Name>
17           <Affiliation></Affiliation>
18     </User>
19      </Origin>
20  <UserText></UserText>
21 </Header>
```

This part of the file includes general information about the version, the author of the file, the affiliation and there is also space for some *UserText* that is intended for any number of text lines that the output tool or the user might find adequate. Then, the EML data file continue with the Declaration Section. For this DD, only one Folder is compulsory and allowed and contain a collection of Variables which are instances of the classed defined in the Data Dictionary definition. The *Variable* can appear in two forms:

- with its own values: the variable will be made to hold data of some shape and type e.g. a vector of string-values or a matrix of integers.
- with references to other variables: the variable will not have any real data. Rather, it will contain references that points at other EML variables in the same file.

Let us see in detail one variable for each type:

```
33 <Variable Name="Horn_Field" Class="Field:Far" ID="1">
34  <Attribute Name="SpaceTypeAxis">Space</Attribute>
35  <Attribute Name="TimeDependency">+j\omegat</Attribute>
36  <Attribute Name="TimeTypeAxis">Frequency</Attribute>
37  <Sizes></Sizes>
38  <Component Reference="Horn_Frequency"/>
39  <Component Reference="Horn_ScanRange_2D"/>
40  <Component Reference="Horn_ProjectionComponents"/>
41  <Component Reference="Horn_PowerNormalisation"/>
42  <Component Reference="Horn_RelativeGainOffset"/>
43    <Component Reference="Horn_PhaseReference"/>
44  <Component Reference="Horn_Directivity"/>
```

```
45 </Variable>
```

This variable is an instance of the Far Field class as specified in the correspondent attribute (at line 33) and is identified by name, which has to be unique in the file, and the ID integer number. Then there are the three compulsory Attributes (lines 34-36) as can be seen in Figure 6.1 with the correspondent values. The next line specify the size that may be empty or containing a sequence of integers. If it is empty, the *Variable* either holds a simple scalar or the variable is a container as in this case. Otherwise the integers tell the size of an n-dimensional array also denoted a Rank n array where each element is a *Component*. Next, there is a list of *Component* which, in this case, are just a reference to other variables of the folder.

Another type of *Variable* is the one containing its own values as:

```
46 <Variable Name="Horn_Frequency" Class="Frequency" ID="2">
47  <Sizes> 2</Sizes>
48  <Component Type="double">
49    <Value> 5 7</Value>
50  </Component>
51 </Variable>
```

where it is easy to recognize the same structure of the opening line as before followed by a non-empty *Sizes* element. In this case the instance of the Frequency class has only one Component with its *Type* and values indicated after the corresponding tag.

The third part of the EML data file is constituted by the Data Section. Usually, the number of data values in a *Variable* is larger than a few so the values will not appear together with the declaration in a *Value* element. Instead, the data will appear in a corresponding element in this section with exactly the same name, equal data types and the attribute *RefID* has to match with the ID attribute in the Declaration Section.

```
111  <Data>
112      <Variable Name="Horn_Directivity" RefID="10">
113          <Component Type="double">
114            <Value> 1.1 1.2 2.1 2.2 21.1 21.2 22.1
```

```
      22.2 31.1 31.2 32.1 32.2 -1.1 -1.2 -2.1
      -2.2 -21.1 -21.2 -22.1 -22.2 -31.1 -31.2
      -32.1 -32.2
127             </Value>
128 </Component>
129 </Variable>
130 </Data>
```

The last part of each EML data file is the Application Data Section. This section is intended for a programs private data that could be anything. This explains why the syntax it is so easy to appear not existing: the maximum freedom is left to the developer. For example the field data dictionary does not include any specification of bearing but some software tools actually used this information. Hence the developers of a specific tool can add all data which are not intended for other tools in this element i.e. a tools private data. In the Example of Appendix A, for simplicity, this section is empty.

```
132 <ApplicationData>
133 </ApplicationData>
```

6.3 Structure Data Dictionary

The EDX dictionaries define exactly and in detail all the elements that will appear in a specific data set that complies with their standard, including all physical and mathematical items and quantities. The Structures Data Dictionary define all is needed to describe a physical structure for electromagnetic modeling. Usually, discretised geometry (mesh) data are stored in long tables where it is not easy to reconstruct quickly and effectively high-level information about the geometry, e.g. the shape of an object, and most of the time, they are not accompanied by various other type of data (e.g. material properties) that are necessary to the design tool. Moreover in practice it is very useful to have multiple description of the same object, for example with different mesh accuracy and a CSG representation, and this is not handled by common geometric formats.

Meeting all these requirements in a robust and consistent way in a data model, calls for a highly structured organization of the information. This explain why a layered structure had been adopted for the Structure DD. Each layer, that will correspond to a *Folder* in the EML file, holds information about a certain type of data elements organized in classes or class hierarchies. The layer that have been created are the following: *Objects, Geometry, Topology, Materials* and *Parameters*. More could be added in the future if need arises.
It is of interest to note that the layered approach is common to several other "rich geometry formats", like many CAD and the Geographical Information System (GIS), in order to associate to purely geometric information additional and extraneous elements.

A much in-depth analysis of this Data Dictionary can be found in next Chapter as, the development and refinement of this DD, is the main goal of this work.

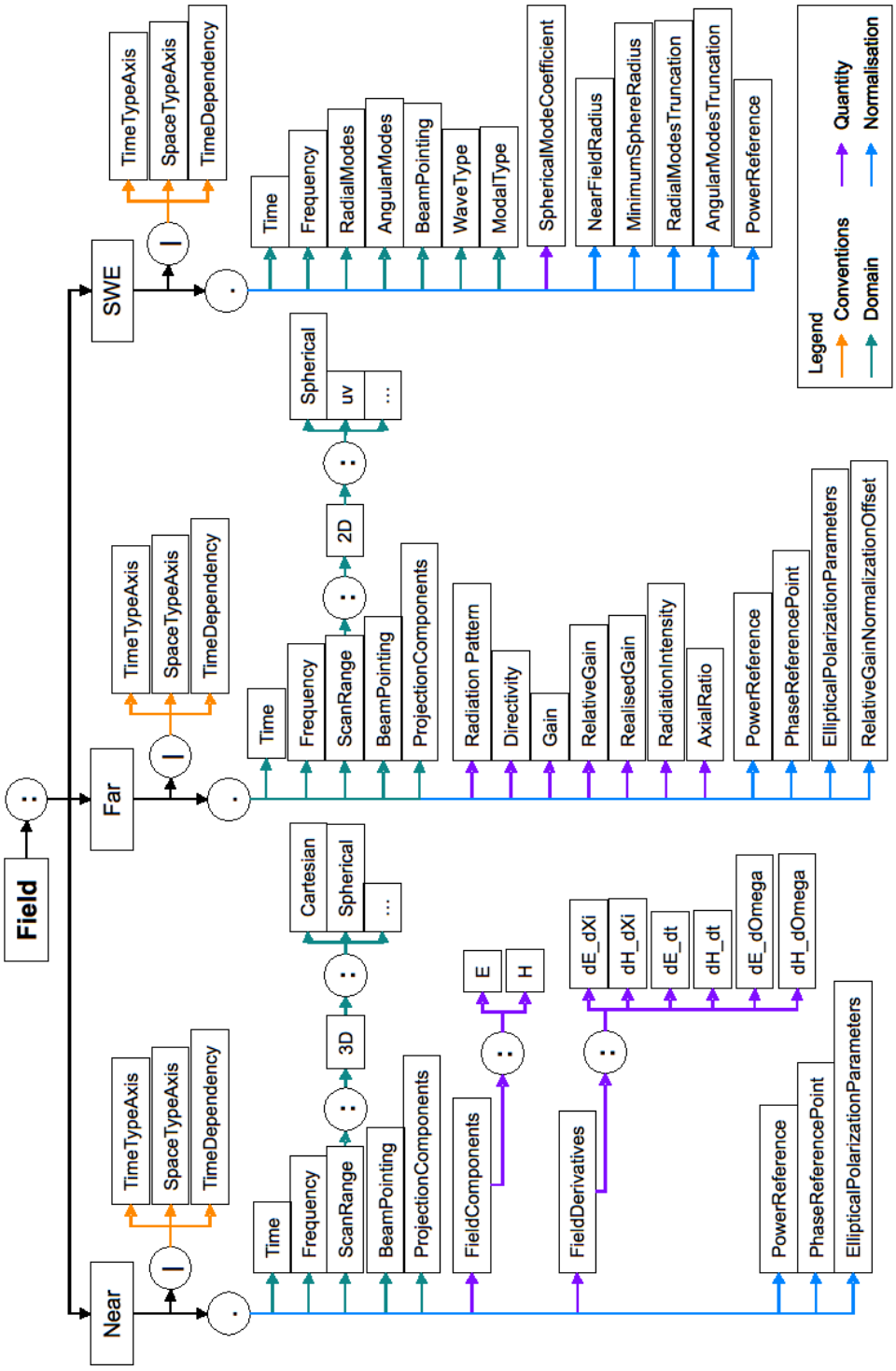


Figure 6.1: Fields Data Dictionary overview [13]

Chapter 7

Structure Data Dictionary

The core of the work was the development of the Structure Data Dictionary intended as an organized collection of all the information necessary to describe a physical structure for electromagnetic modelling.
It is to point out that this collection does not include details about modelling algorithms and their settings nor about the modelling process that generated the data. Such choice is dictated by the need of making the description as general as possible and as independent as possible from modelling methodologies. It is also not a limiting one as EDX offers the possibility to combine multiple data sets, corresponding to different dictionaries, within a single file. Thus allowing for the future exchange or even more complete information in a single container.

The overall structure has been subjected to continuous revision over the study period to reach the final structure shown in Figure 7.1. It is worth noting that this was made possible by another advantage offered by the EDX layered structured and by the modular architecture chosen for this study: changes in the dictionary can be accommodated with very minor changes in the Pyhton modules, most of which are limited to the GEO Modeller. Thus adjustments were possible until the very end of the work.

From the overview, it is already clear that the structure can be divided logically, and therefore practically, in five main *Layers*, namely: Parameters, Materials, Objects, Geometry and Topology. Even if in the Fields Data

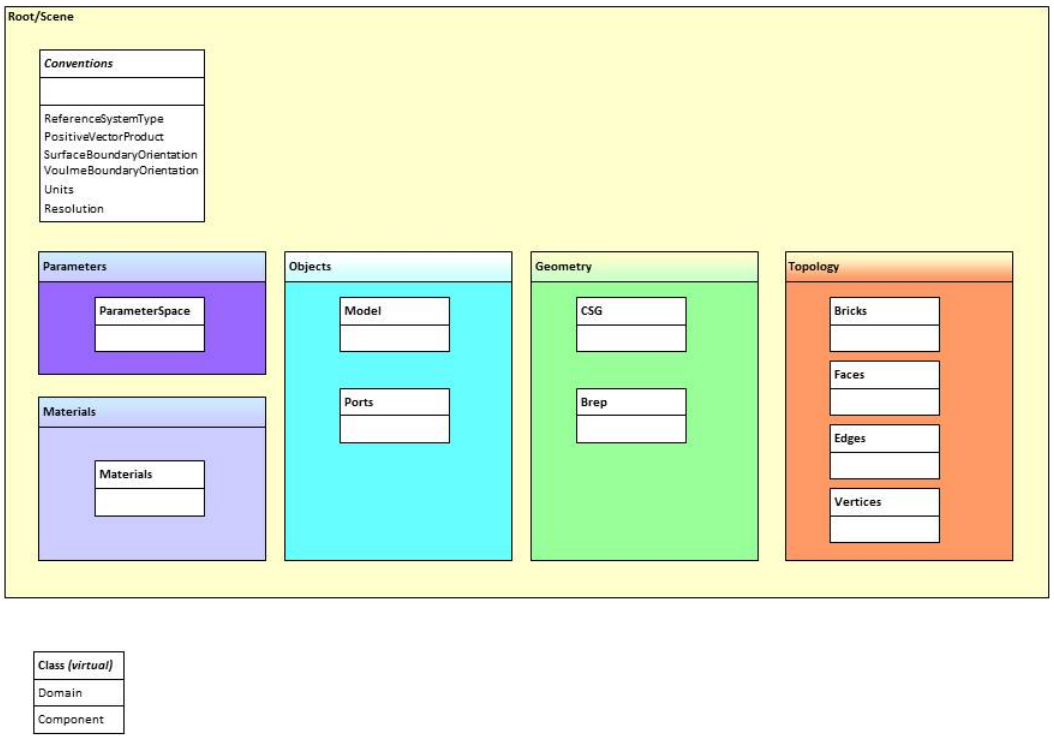


Figure 7.1: Structure Data Dictionary: overview.

Dictionary, they have not been used, the EML syntax makes it possible to use *Subfolders* besides *Folders* and so it seemed reasonable to use just one top-level Folder (*Root*) that contains the class *Conventions* and other five *Subfolders*.

A number of notational conventions are used in Figure 7.1:

- Folders are represented by coloured boxes;
- Classes are mostly shown as blank boxes hosting the names of their attributes ;
- Abstract classes, not supposed to be associated to any variable, are marked by an italicised name;

- Dashed lines with a block arrow represent links among class instances, i.e. the variable of the origin class must host a reference or other pointer to the destination class indicated by the arrow. They do not carry any information about the possible multiplicity of the relation to avoid overloading the picture;

- Solid lines with closed white arrows indicate child-parent relations in the class trees. Note that this implies not only the inheritance of attributes and components, but also the fact that a link to a parent abstract class (dached line) will actually become a reference or pointer to an instance of one of its non-abstract children;

The Objects layer is the starting point for the entire system and outlines the logical hierarchical structure of the whole data set. The *RootElements* class has a list of links to Elements, which are supposed to be the entry points of a hierarchy of other elements composing in more and more detail the structure. It reflects the spontaneous high-level way to describe a complex object starting from the general structure down to the smallest parts constituting the selected element.

The Geometry layer contains the reference systems (*ReferenceSystems* class) and two other sets of classes grouped by the abstract classes *CSGElements* and *BRep*. It is important to note that the geometric information is not only required to specify the overall structure and its components, but also to carry information on the shape of individual mesh elements for non-planar discretisations (e.g. NURBS-based ones).

The Topology layer is composed by the abstract class *Cell* that groups the other four main topologic classes belonging to this layer, while the remaining two (sheets and Siblings). The whole layer is needed for the 4D-BRep description where Brick, Faces, Edges and Vertices (the white boxes), correspond to the usual geometrical elements and offer an entry point from the Geometry layer as well as a way to link back to it, while the yellow classes are added to create the complete 4D-BRep structure.

The Materials layer contains three separate classes each derived from the root abstract class Materials: HomogeneousMaterials, Boundaries and Media. Each class is in fact a list and contains the information required for the description of the electromagnetic properties of three different types of *physical materials*.

The Parameters layer contains a single class ParameterSpace, which instances are lists of parameters with their definition. The symbols defined in these list appear in the Geometry layer in place of numerical quantities allowing for parameterized CSG descriptions. BReps descriptions are instead referred to a single value of all parameters to avoid any geometric and topologic integrity issues. Multiple BReps can be placed in a single data set and associated to individual sets of parameter values.

Finally the Conventions class is in the Root layer and hosts those general items which definition is applicable to the whole.

7.1 Objects Layer

As mentioned before the Object layer is the entry point to the entire system, and consequently, the entire file.
The *RootElement*, that as suggested by its name, is the gateway to the whole description, has a list of links to an abstract class called *Elements* which can be specialized in instances of different classes such as *Structure, Component* or *EquivalentSouces*. However each element derives from the abstract class, three basic information: the name, the reference to an *Aragement* class including a *PlacementRule*, to specify in a more convenient way complex arrangements i.e. arrays and a *ReferencePlacement* pointing to the local reference system, and a link to possible parameters lists.

It is to note that *Structure* class actually group subclasses,in fact there is the need to have quite a few derivative classes to allow an easy and compact definition of more complicated objects.

The *Component* instances have a link to the correspondent material (in the same name Layer) and a connection to *Model* and *Port*.
The link to Model can be multiple to allow the description of several "models" of the same real object without generating fake and confusing new objects for each representation. As natural, every model is composed by parts, each part is an instance of the same name class and has a link to the correspondent geometrical representation whether BRep or CSG. In fact, both possibilities are implemented as forms of the Part class.
The Port class is intended to allow the description of any antenna ports like physical ports or forced excitation, as the two Ports derivative classes show. The link between ports and components is bidirectional in order to make it possible to read both information in any desired order. This is made necessary by the fact that a Port may need to be attached to two different Components or distinct parts of it.

Both *PhysicalPort* and *ForcedExcitation* acquire the attribute name from the higher class and both kind of ports appear to be linked to some part of the boundaries of the object they refer to. The main difference is that the first type (e.g. end-point of a pin or contour of a waveguide) appear to be referred to two parts of the boundaries and so provide the possibility of being defined though a model with two parts or two distinct models, while the second kind (e.g. impressed current) of port is usually link to one piece of the boundary.

Last, the *EquivalentSources* class is the last descendent of the Element class and it is needed to define those electromagnetic fields sources that are not referred to any physical object such as equivalent currents and point sources. They may or may not have a reference to the material depending on the specific case. Moreover, since this class is not link to any Component it has to have a link to a Model to be fully described. It is worth noticing that parts of the model may be common to physical objects since, at the end of the day, it is the same system that is being fully described.

7.2 Topology Layer

The Topology Layer is a simple collection of the six classes required for the description of the 4D-BRep representation discussed in Chapter 5.

The six classes are grouped by the abstract class called *Cell*, which is the entry point to the Layer. The four classes constituting the classical BRep (*Bricks, Faces, Edges* and *Vertices*) contain also a link to the elements in the Geometry Layer (respectively to *Volume, Surface, Curve* and *Points*). To clarify why this link is needed let us consider, for example, an Edge. If it is straight, no other details are needed, otherwise if it is curve its geometry needs to be specify in the correct layer and a link to that description has to be provided.

The other two classes, namely *Sheets* and *Siblings* hosts the other required information to complete the description and their structure is such to support non-manifold topologies as well as volumetric meshes.

Note that the yellow coloured classes (Figure 7.2) is used in the to highlight the fact that several Sheets and Siblings may respectively correspond to a single Face, or Edge.

7.3 Geometry Layer

The Geometry Layer includes all the necessary classes to provide a full geometric description both using CSG or BRep definition. In fact the layer is visibly divided into two main portion: the *CSGElements* and its derivatives and the *BRep* and its descendants.

The first block includes three main realisations which are the *Operator*, that hosts the CSG operators with the correspondent list of primitives it operates on, the *ReferenceSystem* and the *Shapes* which include the four categories of the 3D objects of different dimensionality and referring to the Topology Layer.
A future improvement for the *Operator* class could be to add other operators beside the boolean ones to be able to define easily, for instance, body of revolution or to avoid multiple components and parts to define shapes into

composite, like for juxtaposition.
The *ReferenceSystem* class include the origin, the orientation of the axis and a link to the associated elements which is more natural than having the opposite direction links from the single shapes to the parent reference system that will also force a naming system of the ReferenceSystem instances.

The *BRep* abstract class has three descendents, one for each mesh dimensionality that is *VolumeMesh, SurfaceMesh* and *CurveMesh*. Each of those has a reference to the appropriate topological class and a link to a specification of the mesh parameters. The latter are host in the classes named *VolumeMeshType, SurfaceMeshType* and *CurveMeshType* where the last one is derived by the second one which in turn is derived from the first one. All of those contain only and all the necessary information for the relative dimension mesh.
Note that the indirect link to the topology had been introduced to avoid confusion in the management of multiple copies of the same component. To achieve this the indexing used is at a "layer level" that is to say that the index in the Arrangement in the Object Layer and the index of the link to the Topology in this Layer are equal as so a fully identified portion of the mesh correspond to each copy.

Finally the *BRepApplicability* class host all the relevant details that allow a proper use of the corresponding mesh and to ensure, with a simple check, the validity of the related mesh. This class contains information about the frequency range where the model is valid, the details about the reference values and range for any parameters that may appear in the geometrical description of the corresponding object. Another important information hosts in this class is the reference value (*ParameterReference*)that specifies the actual value used to compute the mesh and the range (*ParameterRange*) where the mesh can be considered applicable.

7.4 Parameters Layer

The Parameters layer is the simplest one that contain just one class called *Parameters Space*. This class is structured as a collection of lists where each

one is independent to avoid possible conflicts in the naming of the parameters.

Each entry of the lists specifies the name, the symbolic value and the so called *range of validity* that means the assumed range of variation of the parameter values, e.g. the one compatible with integrity or other constraints. While at first it seemed to be reasonable, at the end it was decided that no default values are defined here as they are more appropriately indicated directly in the general properties of the BReps elements in the Geometry layer, i.e to the meshes associated to the relevant object (in the Objects layer).
In fact, in the Object Layer there can be any parameters related to any list of the Parameters Layer with an explicit reference to the list; then only the parameter name (i.e. the symbol associated to it) appears in the Geometry Layer to complete the reference, and are implicitly referred to the same list. This apparently awkward solution is made necessary to make sure that the overall integrity can be easily maintained and a single parameter can appear at several places, while still allowing the combination of multiple data sets, a common operation in practical design flows, without requiring complex verifications of the uniqueness of names. Parameter lists are in fact name spaces, with can be seen as declared at a certain level within the Objects hierarchy and are thus visible only within the sub-tree below that point. As the CSG elements in the Geometry layer are directly accessible from the Object elements tree and only from there, the whole structure is actually quite linear to a closer examination.

As said above, the use of multiple parameter lists is required to simplify the combination of information coming from multiple sources and via separate files. As the combination occur at the level of the object tree it is simply required to merge the Parameters layer making sure the $< variable >$ names associated to each list are unique, a task can be easily accomplished also automatically, using a sequential number or the source file name or any other relevant information available at that time, offering a chance to the user to select better names, if so desires. Furthermore, if a ParameterSpace used at a lower level of within the object tree includes a parameter duplicating one defined at an higher level the conflict can be resolved

by simple scoping, as in most programming languages, by making the local definition prevail over the more global one. In the extreme case of a need to use the latter it is always possible to use its fully qualified name, e.g. $< ParameterSpaceName > . < ParmanereName >$.

7.5 Materials Layer

The Material Layer is the most straightforward layer containing three classes namely, *HomogeneousMaterials, Boundaries* and *Media* all derived from the abstract class *Materials*. Each of previous is in fact a, possible long, list of all the necessary information to describe the three type of materials.

The HomogeneousMaterials contain the homogeneous isotropic medium and the conducting surface with a skin depth.

The Boundaries are specified with reflection coefficients, surface impedance and surface admittance.

Finally the Media can be characterized by reflection and transmission matrix or the scattering matrix or impedance matrix or the admittance one. Note that only one description for each material is allowed.

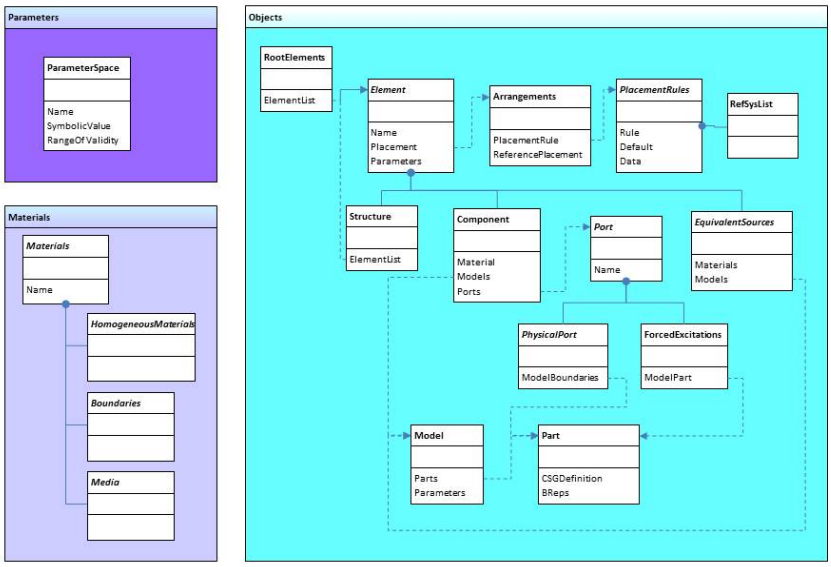


Figure 7.2: Structure Data Dictionary: ParameterSpace, Matierials and Ojcebts layers.

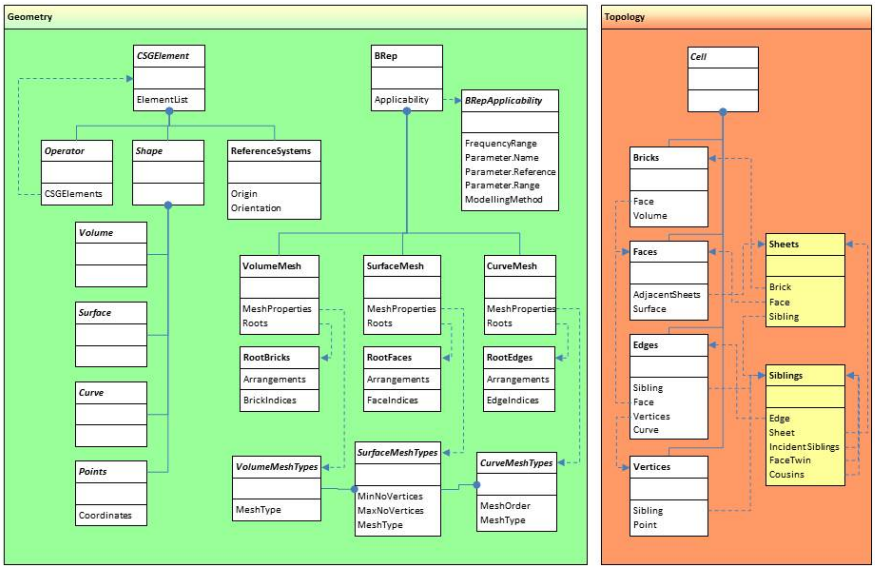


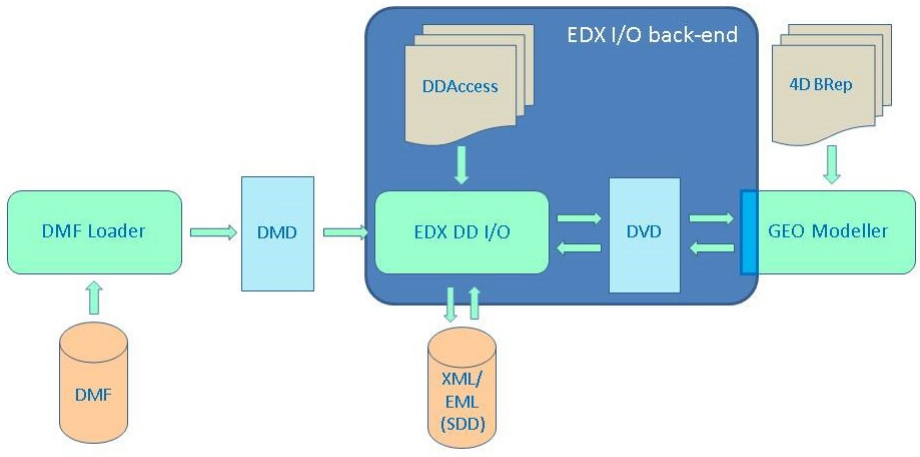
Figure 7.3: Structure Data Dictionary: Geometry and Topology layers.

Chapter 8

Software prototype tool

Starting from the analysis of the antenna design process, paying specific attention to related data model for electromagnetic simulation, a Python-based prototype tool was developed.

The structure has undergone many changes, following the overall iterative approach of the study, until the final version of the program which schematic is shown in Figure 8.1.



DD Data Dictionary (generic)
DMF Data Domain Model File (EDX Data Dictionary formal definition, using the DDL - Dictionary Definition Language)
DMD Data Domain Model Dictionary (Python ordered dictionary encoding the data model)
DVD Data Value Dictionary (Python ordered dictionary encoding the actual data for the selected model)
SDD Structure Data Dictionary

Figure 8.1: Program logical schema.

It is composed by three main logical segments (identified in Figure 8.2 respectively by number 1, 3 and 5):

- EDX S I/O BackEnd: it includes a library of I/O functions to access, both in reading and writing, the XML data files;
- DMF Loader: starting from the formal definition of the language (coded in the Dictionary Declaration Language [31]) it checks the lexicon and the syntax and build the Data Model Dictionary (DMD).
- GEO Modeler: it includes a library of functions to assemble the geometrical and topological information and create the Data Values Dictionary (DVD).

Each portion of the program can also be used independently as stand alone tool. They have indeed been developed at different times and integrated only in the last stages of the study.
The coordination and interactions between the different modules are managed by a single main script that inherently has many degree of freedom to give the user the possibility to employ it fully.

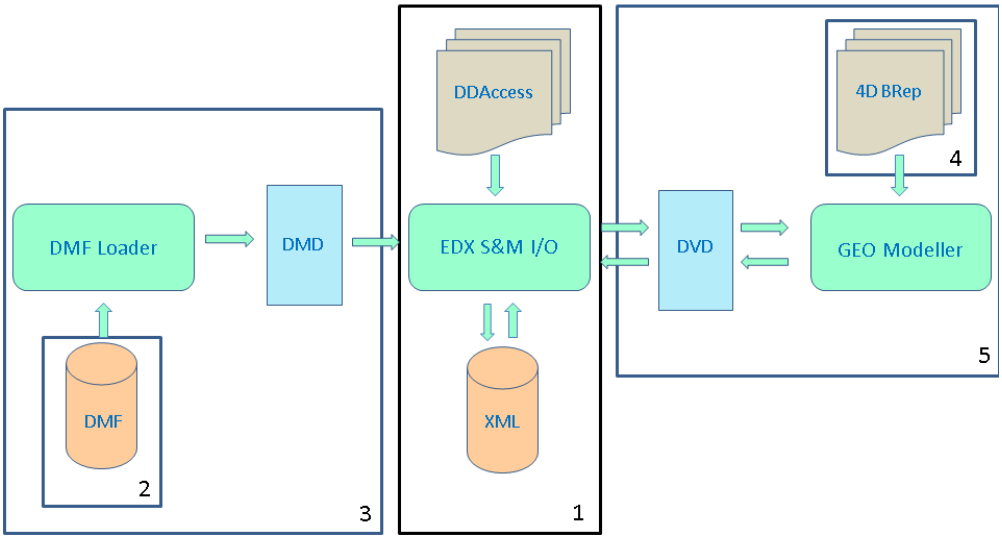


Figure 8.2: Program logical schema: step by step.

8.1 EDX I/O BackEnd

Due to the natural hierarchical structure of the XML data file (an example can be found in Appendix A) it had stand to reason to map the file into a tree data type. The *lxml.etree* toolkit [37] was used to do so, where the root of the tree data type has been mapped to be the beginning of the file itself and each section is mapped as follow:

```
etree.root = <Element {http://www.edi-forum.org}EDIFile>
etree.root[1] = <Element {http://www.edi-forum.org}Header>
etree.root[3] = <Element {http://www.edi-forum.org}Declarations>
etree.root[5] = <Element {http://www.edi-forum.org}Data>
etree.root[7] = <Element{http://www.edi-forum.org}ApplicationData>
```

The other elements attached to the root (etree.root[0], etree.root[2], etree.root[4], etree.root[6]) are just place-holder, necessary for the proper handling of the typical EML file, that contains the comment line including the name of the section and are irrelevant for the purpose of the study.

Initially this library was divided in four classes, one for each section, including all the functions operating on that part of the file. After some practise, it has been seen that this initial structure was somewhat redundant and all functions could be rewritten to be independent from the section of belonging and so have been grouped in one single class.

The latest version of the program includes 43 functions: 4 to assign the elements of each section to the correspondent node of the tree data structure, 17 for reading the files, 21 for writing it, one to write the etree to a file.
The presence of the other allows for a greater flexibility and the user can perform all the desirable operation on the data files, possibly calling the functions from a Python interpreter command line, e.g. for debugging purposes. In fact the main reason for their development has been the testing of the module accompanied by the which to offer a complete set of functions to future developers.
The Application Programming Interface (API) to the tool supposed to receive or generated the actual data, the GEO Modeler in our case, is provided by

a subset of functions indicated with "Y" (Yes) in the second column of the Table 6.1.

In the same table a complete list of functions, used at the moment in the final integrated tool, can be found.

Table 8.1: Complete list of I/O functions.

Function Name	**API**	**Description**
Header	N	Assign the Header elements to the etree
AssignHeaderElement	N	Assign or change one Header element
Declaration	Y	Assign the Declaration elements to the etree
Data	Y	Assign the Data elements to the etree
ReadHeader	N	Return the information of the header
Folders	N	Return all the folders info of the DS
FirstLevelFolder s	N	Return a list of the folders of the DS
SecondLevelFolders	N	Return a list of the subfolders of the DS
SearchVariable	N	Look for a variable in the DS
SearchFolder	N	Look for a folder in the DS
Variables	N	Return a list of the variables of the DS
Attributes	Y	Return a list of the attributes of the folder passed in input
Domains	Y	Return a list of the domains of the folder passed in input
Names	Y	Return a list of the names of the folder passed in input
Sizes	Y	Return a list of the sizes of the variables of the folder passed in input
Values	Y	Return a list of the values of the variables of the folder passed in input
getDD	N	Return the Data Dictionary used in the file
CreateVariableList	N	Return a list of the variables contained in the Folder passed as input

Continued on next page

Table 8.1 – continued from previous page

Function Name	API	Description
DataVariables	N	Return a list of the variables that have data in the Data Section
ReadData	Y	Return a list of the data in the Data Section
ReadApplicationData	N	Return the Application Data
WriteVersion	N	Write the version in the Header section
WriteFormat	N	Write the format in the Header section
WriteDateTime	N	Write the date and time in the Header section
WriteToolName	N	Write the tool name in the Header section
WriteToolVersion	N	Write the tool version in the Header section
WriteProject	N	Write the project name in the Header section
WriteUserName	N	Write the user name in the Header section
WriteAffiliation	N	Write the user affiliation in the Header section
WriteUserText	N	Write the user text in the Header section
SetFolder	Y	Set a folder in the DS
SetVariable	Y	Set a variable in the DS
SetAttribute	Y	Set an attribute in the DS
SetValue	Y	Set the value of a variable in the DS
SetSize	Y	Set the size of a variable in the DS
SetType	Y	Set the type of a variable in the DS
SetUnits	Y	Set the units of a variable in the DS
SetDomain	Y	Set the domain of a variable in the DS
WriteComponent	Y	Set a component of a variable in the DS
SetStructure	Y	Set the structure of a variable in the DS
WriteData	Y	Write the data in the Data section
WriteApplicationData	N	Write the application data in the Application Data section
WriteToFile	Y	Write to etree to the file passed in input

8.2 DMF Loader

This portion of the program includes the Data Model File (DMF) module (identified in Figure 8.2 with the number 2) which is a lexical and syntactical analyser based on PLY which is a Python implementation of LEX/YACC, respectively a lexical analyser and a compilers compiler evolved over many years in open-source UNIX and LINUX world.

This module takes as input the formal definition of an EDD and first check the lexicon of the file and then the syntax. If no error are found it produces the Data Model Dictionary (DMD) which is a Python ordered dictionary (called Dict) containing all useful information extracted from the EDD that had been read and encoded in such a way to make its use as simple and straightforward as possible in the core of the EDX I/O module.

For the lexical analyzer a list of allowed tokens are defined using the BNF (Backus-Naur Form). They are grouped in:

- ID: $[+a-zA-Z\_][a-zA-Z\_ : //0-9]^*[0-9a-zA-Z]$
- VALUE: $([-]^*[0-9][0-9]^*[\ .0-9]^*)^+$
- reserved: a dictionary of key words that has to be identified as is, without being confused with the previous categories
- newline: $\backslash n^+$

After the definition of the tokens, the lexer core is built. Then, YACC is set with all the necessary productions to be flexible enough to recognize all the plausible dictionaries derived from the Data Dictionary Declaration Language paying particular attention to disallow all the other constructions.

To give an example of the logical operation of the YACC let us consider the following excerpt from a DDL file:

```
([ new ] | override)
*{attribute <attribute name> : <value>*{,<value>}}
*{domain <domain name> reference <class name>}
[structure (CartesianProduct | ListOfTuples)]
[
```

```
   units <unit symbol>
   type <type name>
   size <number> *{,<number>}
  |
   +{ component <component name>
    (
      units <unit symbol>
      type <type name>
      [size <number>*{,<number>}]
      [association<class name>]
    |
      reference <class name>
    )
    }
  ]
```

This portion is translated into successive productions, where the words in capital letters are the reserved words or other previously defined tokens (terminal symbols), while the remaining ones represent a link to other productions.

In correspondence of each definition it is compulsory to define an action that the YACC has to perform at the end of a rule. Two main kind of rules had been used: if no real action is needed a "pass" operation is set, otherwise the *Dict* dictionary entry is created and written on a file (called *dd*).

```
    <<NewClassDeclaration : StatusDeclaration
     ZeroOrMoreAttributes ZeroOrMoreDomains
     StructureDeclaration ComponentsDeclaration>>
     pass

def  p_StatusDeclaration(t):
     <<StatusDeclaration : NEW
                         | OVERRIDE
                         | empty>>
     pass
```

```
def p_ZeroOrMoreAttributes(t):
    <<ZeroOrMoreAttributes : ZeroOrMoreAttributes
                           AttributeDeclaration
                           | AttributeDeclaration
                           | empty>>
    pass

def p_AttributeDeclaration(t):
    <<AttributeDeclaration : ATTRIBUTE ID ':' ID>>
    a = FolderName+'|'+Level+'|'+'_AT_'+'|'+VarClass+'|'+t[2]
    dd.write("Dict['%s']='%s' \n" %(a, t[4]))

def p_ZeroOrMoreDomains(t):
    <<ZeroOrMoreDomains : ZeroOrMoreDomains
                          DomainDeclaration
                        | empty>>
    pass

def p_DomainDeclaration(t):
    <<DomainDeclaration : DOMAIN ID REFERENCE ID
                        | DOMAIN REFERENCE ID>>
    if len(t) > 4:
        a = FolderName+'|'+Level+'|'+'_DO_'+'|'+VarClass+
        +'|'+t[2]
        b = t[4]
        dd.write("Dict['%s']='%s' \n" %(a, t[4]))
    else:
        a = FolderName+'|'+Level+'|'+'_DO_'+'|'+VarClass+
        +'|'+'Integer'
        b= t[3]
        dd.write("Dict['%s']='%s' \n" %(a, t[3]))

def p_StructureDeclaration(t):
    <<StructureDeclaration : STRUCTURE StructureQualifiers
                           | empty>>
```

```
    pass

def p_StructureQualifiers(t):
    <<StructureQualifiers :  CARTESIANPRODUCT
                           | LISTOFTUPLES>>
    TypeOfStructure = t[1]
    a = FolderName+'|'+Level+'|'+'_ST_'+'|'+VarClass
    b = TypeOfStructure
    dd.write("Dict['%s']='%s' \n" %(a, t[1]))

def p_ComponentsDeclaration(t):
    <<ComponentsDeclaration : Units Type Size
                            | OneOrMoreComponents
                            | empty>>
    pass
```

The keys of the pairs to be entered into the Dict dictionary are then composed, which can be described as:

key := FolderName | FolderType |_Identifier_| ClassName | (opts)

- FolderName: the name of the folder (or subfolder) as defined in the corresponding EDD;
- FolderType: can be _FO_ or _SF_ that respectively mean "Folder" and "Subfolder";
- Identifier: two character, preceded and followed by an underscore, identifying the corresponding XML tag (see Table 6.2);
- ClassName: the name of the corresponding class as defined in the EDD;
- (opts): if the considered element is a Component then here there will be a counter to univocally distinguish and group the different ones for the same class.

Is it to be noted that the entry corresponding to a definition of a new Folder has a different structure, that is to say:

key := DataDictionaryName | FolderName | FolderType value := FolderName

with similar meaning as above.

Finally, the values of the entries are default values that will be substituted by the actual data via the calls to the API functions in the EDX I/O BackEnd, i.e. through the GEOModeller in our case. For testing purposes they were mostly inserted by hand modifying the Python script where the dictionary is actually written.

8.3 GEO Modeller

This module has been developed as last and is identified by number 5 in Figure 8.2. The *4DBRep* (Figure 8.2 n. 4) is the core of the module and consist of a library of topological and geometrical functions that interact with the rest of the Geo Modeller with two methods, one for reading and one for writing.

The first step, to the development of this library, had been the Python implementation of the BRep representation, that is to say that all the topological relations among bricks (B), faces (F), edges (E) and vertices (V), were created. At a later stage, the 4DBRep representation was completed adding the relationship concerning the sheets (H) and the siblings (I).
At the beginning also the nodes (N) were introduced but then, after some trial examples, it resulted that the nodes were somewhat redundant and all functions could be rewritten to be independent from them and all the needed relations could be reconstruct from the other elements. The structures of the 4DBRep library is shown in Figure 8.3 where the following relations were implemented:

Table 8.2: Complete list of Identifier for DMD.

Relation	**Name used**	**Description**
HB	HB	to each sheet is associated the corresponding brick
BF*	BF	for each brick a face in linked
FH	FH	each face is linked to the two corresponding sheets
HF	HF	each sheet has a reference to the face
EF*	EF	edges are connected to one face
HI*	HI	each sheet has a link to one of its sibling
IH	IH	siblings has a reference to the belonging sheet
IE	IE	each sibling refer to the corresponding edge
EI*	EI	edges have link to just one of their sibling

Continued on next page

Table 8.2 – continued from previous page

Relation	Name used	Description
II full-line	II	for each sibling is specify the previous and following sibling according to the corresponding sheet
II dot-line	IC	siblings have a reference to the corresponding one that belong to another brick
EV	EV	for each each are individuated the two end vertices
VI*	VI	each vertex has a reference to one of the sibling

Note that the star near some associations in the previous table means that it is a partial relation.

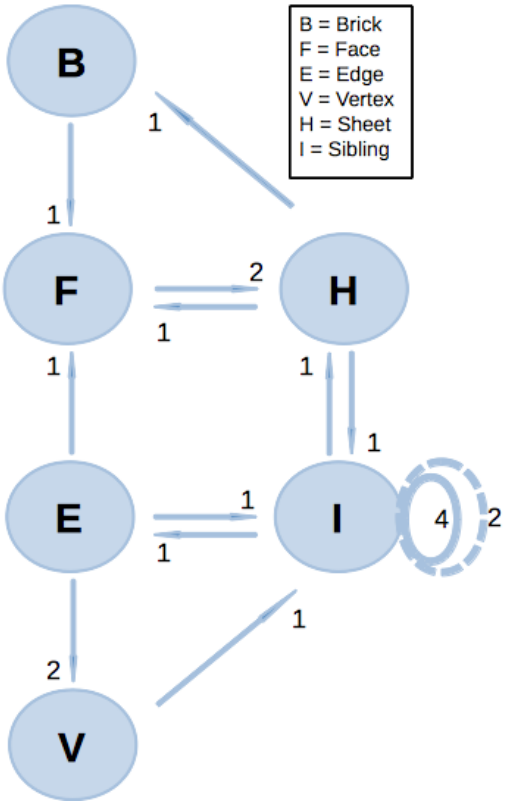


Figure 8.3: Structure of 4DBrep.

It is to highlight that these relations are required and sufficient to rebuilt all the others starting for any point to any other, as proved in Figure 8.5. Beside that, also other relations were implemented such as: *FE* (face to edge), *faceTwin*, for each sibling it is identify the correspondent sibling belonging to another brick and *adjacentSheets* that link the two sheets of the same face.

Finally, the points, which are nothing more than the coordinates of the vertices, are read and written by the same library even if they logically belong to the Geometrical Layer instead of the Topological Layer as all the others, beforehand mentioned.

To check the completeness and the correctness of the library several examples and tests were done, starting with simple shapes such as boxes and plates up to the more complex examples whose description can be found in Chapter 8. The general procedure is summarised in Figure 8.4: an empty model is created, then a shape is generated and added to the model. Eventually the model is plotted and the corresponding EDX data file is written. Afterwards the data file is read, the model is rebuilt and is it plotted: if the initial shape and this last one displayed are the same we are reasonably sure that the data structure is complete and correct.

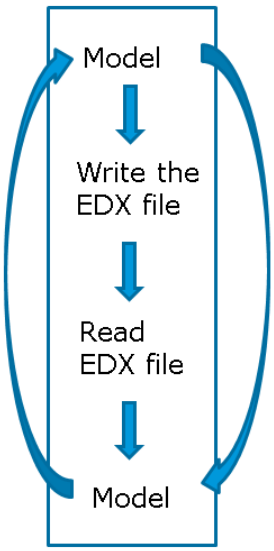


Figure 8.4: Basic flow chart.

To prove that, it is enough to reconstruct the information (vertices and sheets) needed for plotting, both in writing and in reading, from the data structure built. Two dedicated functions were implemented for that purpose:

- getVertices: that through the vertices extracts the points;
- getHV: that rebuilt a matrix containing the coordinates of the points for each sheet. It can be sum up with the following reconstruction algorithm:

$$HV = HI^* * [II] * IE * EV$$

  where once again the star stands for partial and the square parenthesis stand for the iteration over the usual BRep structure.

It is worth to stress that in this module *writing* and *reading* actually mean respectively to write and read the Data Value Dictionary (DVD) that is a collection of dictionaries, one for each Layer, containing the real values that will be passed to the EDX I/O back end to *write* or *read* the file.

While the Topology dictionary is fully fill up with the actual values automatically from the 4DBRep library, the other dictionary are at the moment filled up by handwriting the dictionaries using another script of the module called "otherFolders". This part is surely open to future improvements.

rev	Inverse relation
*	Partial encoding
■	Full encoding
‖	Encoding by direct concatenation
°	Relation restricted to a brick
□	Relation restricted to a face

ˆ	Relation restricted to an edge
^	Relation restricted to an edge
[ ]	HIN winged edge relation
{ }	Involved avatar relation
·	Concatenation of relations
$( )^a$	Repeated application of a relation

Relation	Encoding	Reconstruction algorithm for full/brick relation		Cost	Possible alternatives
Cell relations					
BB		BB = BF*·FH·[HH]·HB		O(n)	BB = BH·HB = BF·FB
BF	*	BF = BF*·FH*·[HH]·HF		O(n)	
Rev	‖	FB = FH·HB		O(1)	
BE		BE = BF*·FE*·EI*·[II]·IE		O(n)	BE = BI·IE = BF·FE
Rev		EB = EI*·{Iˆ1}·IH ·HB	E°B* = EI·IH·HB	O(np)	
BV		BV = BF*·FE*·EI*·[II]·IN·NV		O(n)	BV = BF·FV
Rev		VB = VN*·NI*·{Iˆ1}·[II]·IH·HB	V°B* = VN*·NI*·IH·HB	O(np)	
FF		FF = FH*·HI*·{Iˆ1}·[II]·IH·HF	F°F = FH*·HI*·[II]·IH·HF	O(np)	
FE	(*)	FE = FH*·HI*·[II]·IE	FE* = FH*·HI*·IE	O(n)	
Rev	*	EF = EI·{Iˆ1}·IH·HF	E°F = EI·[IH]·HF	O(np)	
FV		FV = FH*·HI*·[II]·IN·NV		O(n)	
Rev		VF = VN*·NI*·{Iˆ1}·[II]·IH·HF	V°F = VN·[NH]·HF	O(np)	
EE		EE = EI·{Iˆ1}·[II]·IE	E°E = EI·[II]·IE	O(np)	
EV	■			O(1)	
Rev		VE = VN*·{NˆN}·NI·IE*	V°E = VN*·[NI]·IE*	O(np)	
VV		VV = VN*·{NˆN}·NV	V°V = VN*·[NN]·NV	O(np)	VV = VN·NV
Cell-avatar relations					
BH	‖	BH = BF*·FH		O(1)	
Rev	■			O(1)	
BI		BI = BF*·FE*·EI*·[II]		O(n)	
Rev	(°)	IB = I°B = IH·HB		O(1)	
BN		BN = BF*·FE*·EI*·[IN]		O(n)	
Rev	(°)	NB = N°B = NI·IH·HB		O(n)	
FH	*	FH = FH*·HˆH		O(n)	
Rev	■			O(1)	
FI		FI = FH*·HˆH·[HI]	F°I = FH*·[HI]	O(n)	FI = FH·[HI]
Rev	(°)	IF = I°F = IH·HF		O(1)	
FN		FN = FH*·HˆH·[HI]·IN	F°N = FH*·[HI]·IN	O(n)	FN = FI·IN
Rev	(°)	NF = N°F = [NH]·HF		O(n)	
EI	*	EI = EI*·{Iˆ1}		O(np)	
Rev	■			O(1)	
EN		EN = EI*·{Iˆ1}·IN	E°N = EI*·[II]·IN	O(np)	EN = EI·IN
Rev	(°)	NE = N°E = NI·[II]·IE		O(n)	
VN	*	VN = VN*·{NˆN}	V°N = VN*·[NN]	O(np)	
Rev	■			O(1)	
Avatar relations					
HˆH	‖	HˆH = HF·FH		O(1)	
Iˆ1		$Iˆ1 = (\{(IH·HˆH·HI^*·[II])_{IE_0=IE_k \cup II}\}_{IL_{0k}\neq IL_{0l}})^p$		O(np)	Iˆ1 = {IN·NˆN·[NI]}$_{IL_{0k}\neq IL_{0l}}$: O(p), if ∃NˆN Iˆ1 = (Iˆ1*)ᵖ: O(p), if Iˆ1* is full
NˆN		$NˆN = [(\{(NI^*·IH·HˆH·HI^*·[II])_{IE_0=IE_k \cup II}\}_{IL_{0k}\neq IL_{0l}})^p·IN]_{NN_{0k}\neq NN_{0l}}$		O(np)	NˆN = (NI*·Iˆ1·IN)$_{NN_{0k}\neq NN_{0l}}$: O(p), if ∃Iˆ1 NˆN = (NˆN*)ᵖ: O(p), if ∃NˆN*

Relation	Encoding	Reconstruction algorithm for full/brick relation	Cost	Possible alternatives
Brick WE relations				
HH		HH = HI*·II·IH	O(n)	HH = HI·IH
HI	*	HI = HI*·II	O(n)	
Rev	■		O(1)	
HN		HN = HI*·II·IN	O(n)	HH = HI·IN
Rev		NH = NI*·II·IH	O(n)	NH = NI·IH
II	*	II = $(II^*)^n$	O(n)	
IN	■		O(1)	
Rev		NI = NI*·II	O(n)	
NN		NN = NI*·II·IN	O(n)	NN = NI·IN

Figure 8.5: Reconstruction of topological relations in a reduced 4D-BRep with partial encoding.

Chapter 9

Results

A preliminary, but necessary, result is the deep analysis and understanding of the whole design process of the antennas the related problems and needs. This involved a not negligible critical study on the copious literature on the arguments and a direct comparison with international antenna specialists.

Moreover, since this work is inserted in the much bigger frame of the EDX, a substantial work on the project background had to be performed. This entails a deep, both theoretical and practical, understanding of the Fields Data Dictionary and on Electromagnetic Mark-Up Language (EML).

However the most relevant theoretical results have been achieved with the development of the structured data model constituting the Structure Data Dictionary detailed in Chapter 6.

About the practical result, the most tangible one is the development of a Python-based prototype tool to manage the physical structure data model with particular attention to the geometrical and topological information. The final structure of the program is brought back in Figure 9.1 (see also Chapter 7).

As described in detail in Chapter 6, the tool is logically divided into three main parts:

- EDX I/O back-end: it contains a complete sets of functions to fully access the EML data file.
- DMF Loader: it is constituted by an implementation of a lexical (LEX)

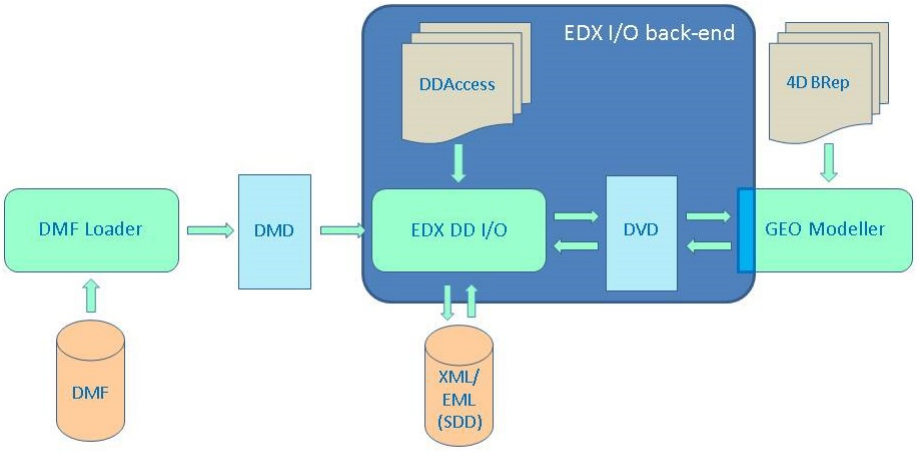


Figure 9.1: Program logical schema.

and syntactical (YACC) analyser. It is able to read the formal definition of the EDX Language, check the lexicon, then the syntax and if no errors are found it create the DMD which is a ordered dictionary containing all the information on the data model needed to fill up the XML file.

- Geo Modeller: it includes a library of functions to assemble the geometrical and topological information and encode them into the DVD which is a dictionary containing the actual values that have to be written to the EML file or that had been read from it.

The coordination and interaction between different scripts are managed by a single main script that inherently has many degree of freedom to give the user the possibility to employ it fully.
It is to note that each portion can also be used as stand alone program. Just to give an idea the tool is made by eight scripts for a total just below 3000 lines of code half of which constituting the 4D-BRep script.

At first the prototype had been applied to create simple example to it-

eratively validate and consolidate the Structure Data Dictionary that have been produced and refined at the same time (a complete description of it can be found in Chapter 6). Then the program, corrected, completed and consolidated in turn, had been applied to produce a number of more complex and complete examples.

Next section describes in detail a simple example and later on some more complex examples will be outlined.

9.1 Example 1

Let us consider three simple shapes: two boxes(depth=1, length=3, height=3) and one square plate (side=2) disposed as shown in Figure 9.2.

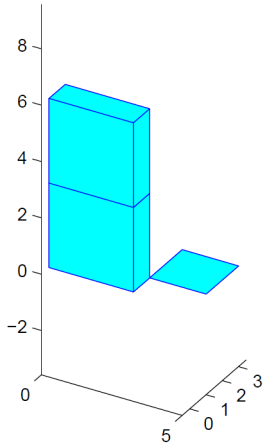


Figure 9.2: Configuration example 1.

The first operation is to describe the objects we want to write in the EML file in the function called *GeoModeller* in the Main script. In this specific case:

```
1 def GeoModeller(self):
2     model = Model()
3     box_1 = box(3,1,3, [0,0,0], np.eye(3))
4     model.addEntity(box_1)
5     box_2 = box(3,1,3, [0,0,3], np.eye(3))
6     model.addEntity(box_2)
```

```
7       model.unifyWE()
8       Plate = plate(np.array([[2,1,0], [4,1,0], [4,3,0],
            [2,3,0]]), [1,0,0], np.eye(3))
9       model.addEntity(Plate)
10      fakeShape = model.unifyWE()
11
12      fakeModel = Model([])
13      fakeModel.addEntity(fakeShape)
14      fakeModel.drawCollection()
15      fakeModel.writeDVD()
```

The general procedure, regardless the specific objects to describe, it to create an empty model (line 2), create the desired objects using the different types of shapes available in the *Shape* class of the 4DBRep script, two by two. It means that after having defined two shapes, if they have something in common with other figures we want to represent it is necessary to merge the firsts into a new shape before adding others (line 7, 10).
When merging shapes there are three possible behaviour:

- the shapes have different dimensionalities: only the points, if there are any shared ones, are actually merged;
- the shapes have the same dimensionality but do not share anything: the corresponding 4D winged edges are just concatenated and the indices updated;
- the shapes have the same dimensionality and share something: the corresponding 4D winged edges are actually merged and a new shape is returned.

So what happen at line 7 is the third possibility, while at line 10 is the first type of operation that is performed. The reason why the product of line 10 is called "fakeShape" is because is not actually a new shape (it is not memorized anywhere) and the same would have happened in the case of another shape of the same dimensionality of the first two but isolated from them.

Afterwards a new empty model is created (line 12) with the only purpose of adding the "fakeShape" to be able to draw it and write the corresponding DVD.

The function *drawCollection()* (line 14) of the class Model simply call the function "Shape.draw" on each shape contained in the model.
The drawing function reconstruct the information needed for the plot (vertices and sheets) by extracting them from a chain of relations as described in Chapter 6. The plot obtained is in Figure 9.3 and represent a first confirm that what will be written in the EML file is, actually, what we wanted.

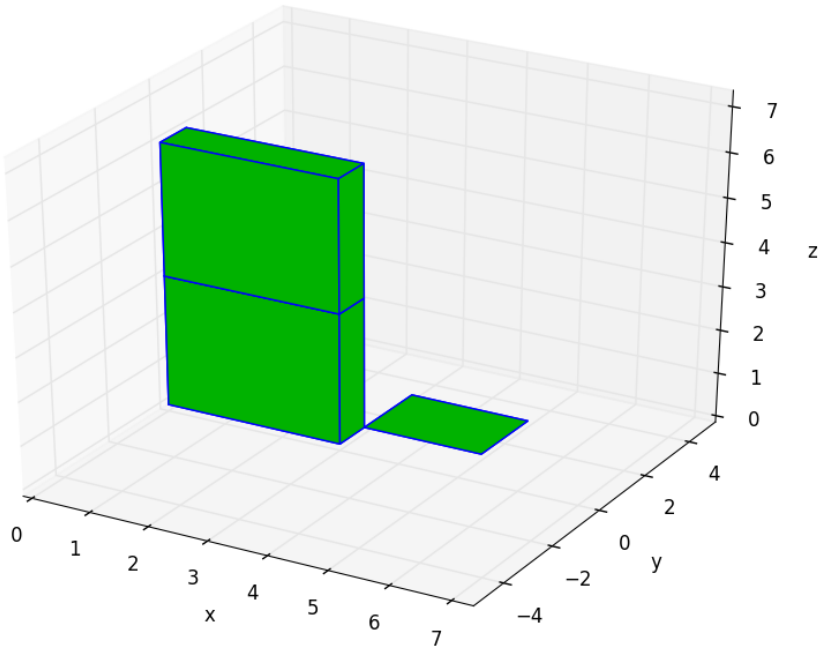


Figure 9.3: Example 1 plot.

Finally, the Data Value Dictionary (DVD) is written (line 15). This function, is actually a list of calls to the function *write* in the WEdge class that receive a number of parameters, including a reference to the Dictionary where to write and the composed key that has to be added.

At this point the Topology dictionary and part of the Geometry one are filled up. To complete the DVD, that includes also the Root, the Objects, the Materials and the ParateterSpace dictionaries, another script, called *otherFolders* is used.
This last needs to manually introduce the correct values on the correspondent keys which information are required to appear in the EDX file.

To check the correctness of the data file created, the most efficient and simple way is to read the file, extract all the information contained and plot them: if the image displayed when writing and the one rebuilt when reading are the same we can reasonably be sure that the data file is correct.

To be more specific, the *readGeoModeller()* function, in the main script, create a new model containing an empty winged edge only. Then the *read* function of the WEdge class is called a number of times looking for all the relations and associating them to the correspondent winged edge information. Then the new shape individuated by the winged edge generated in this manner is added to the model and this last is plotted using the same *drawColletion()* function used before.

The EDX data file for this example is shown below. It can be clearly seen the division of the file in the four main sections: Header (lines 2-21), Declaration (lines 22-315), Data (lines 316-364) and Application Data (lines 365-367). The Data section usually is the most substantial part of the file potentially containing huge amount of data.

In the Declaration section the outline of the Structure Data Dictionary can be point out:

- Root Folder (lines 1-314): contains all the other layers and the *Conventions* class (lines 25-32) with the correspondent attributes as set in the Data Dictionary Definition.

- Object Folder (lines 33-122): comprehends the *Structure* class (line 41) which has *Size*=3 and in the values host the name of the three objects, *Box_1*, *Box_2* and *Plate*. These components are described respectively in line 50, 56 and 62 in same names variables instances of the *Components* class. Each of those refer to a *Model* instance (*MyModel_1* (line 76), *MyModel_2* (line 82) and *MyModel_3* (line 88)). Every of the last variables, in this case, has just one *Part* reference which are realize in the variables at lines 95, 104 and 113. Each *Parts* instance variables hosts also two references about their geometrical representa-

tion that is to say one reference to the *CSGDefinition* which value is SolidBox for *Box_1* and *Box_2* and RectangularPlate for the third shape, and the other reference to the *BRep* representation respectively *Box_1Mesh, Box_2Mesh* and *Plate_Mesh*.

- Geometry Folder (lines 123-199): contains in particular the *Points* class (lines 127.133) which size is 20 and has just one component *Coordiantes* each one of size of 3, so in total there will be 60 values in the correspondent variable in the Data Section (line 320).

- Topology Folder (lines 200-256): hosts the seven classes that compose the 4DBrep representation with the respective sizes and values. If the numbers of values is greater than twenty, than they are written in the Data Section in a variable that has the same name and the same reference ID.

- Parameters Folder (lines 257-264): no parameters have been used in this example, so the layer appear to have no values in its variables.

- Materials Folder (lines 265-313): the only material used is Aluminium (lines 266-276) as the variable at line 266 point out. This material, belonging to the *ConductingSurface* class, is described by its electrical conductivity and magnetic permeability and the frequency of characterization is reported.

```
1  <EDIFile xmlns="http://www.edi-forum.org" xmlns:xsi="http:
       //www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.edi-forum.org edi.xsd">
2    <!-- === Header section === -->
3    <Header>
4      <Stamps>
5        <Version>EDI Version 1.00.00</Version>
6        <Format>XML</Format>
7        <DateTime/>
8      </Stamps>
9      <Origin>
10       <Tool>
```

```
11          <Name/>
12          <Version/>
13        </Tool>
14        <Project/>
15        <User>
16          <Name>Francesca Rossi</Name>
17          <Affiliation/>
18        </User>
19      </Origin>
20      <UserText/>
21    </Header>
22    <!-- === Declarations section === -->
23    <Declarations>
24      <Folder Name="Root" ID="0" DataDictionary="
            StructureAndMeshDataDictionary">
25        <Variable Name="Conventions" Class="Conventions" ID="
            1">
26          <Attribute Name="SystemReferenceType">Cartesian</
              Attribute>
27          <Attribute Name="PositiveVectorProduct">LeftHand</
              Attribute>
28          <Attribute Name="SurfaceBoundaryOrientation">
              CounterClockWise</Attribute>
29          <Attribute Name="volumeBoundaryOrientation">Outward
              </Attribute>
30          <Attribute Name="Units">mm</Attribute>
31          <Attribute Name="Resolution">Double</Attribute>
32        </Variable>
33        <Folder Name="Objects" ID="1">
34          <Variable Name="RootElements" Class="RootElements"
              ID="2">
35            <Component Name="Element" Reference="Element"/>
36          </Variable>
37          <Variable Name="Arrangements" Class="Arrangements"
              ID="3">
38            <Component Name="PlacementRule"/>
39            <Component Name="ReferencePlacement" Reference="
                ReferenceSystem"/>
40          </Variable>
```

```
41          <Variable Name="MyStructure" Class="Structure" ID="
                4">
42            <Component Name="ElementList" Type="String"
                 Reference="ParametersSpace">
43              <Size> 3</Size>
44              <Value> Box_1, Box_2, Plate</Value>
45            </Component>
46            <Attribute Name="Name">MyStructure</Attribute>
47            <Component Name="Placement"/>
48            <Component Name="Parameters"/>
49          </Variable>
50          <Variable Name="Box_1" Class="Component" ID="5">
51            <Component Name="Models" Type="String" Reference=
                 "Model">
52              <Size> 1</Size>
53              <Value> MyModel_1</Value>
54            </Component>
55          </Variable>
56          <Variable Name="Box_2" Class="Component" ID="6">
57            <Component Name="Models" Type="String" Reference=
                 "Model">
58              <Size> 1</Size>
59              <Value> MyModel_2</Value>
60            </Component>
61          </Variable>
62          <Variable Name="Plate" Class="Component" ID="7">
63            <Component Name="Models" Type="String" Reference=
                 "Port">
64              <Size> 1</Size>
65              <Value> MyModel_3</Value>
66            </Component>
67            <Component Name="Placement"/>
68            <Component Name="Parameters"/>
69            <Component Name="Material"/>
70            <Component Name="Ports"/>
71          </Variable>
72          <Variable Name="EquivalentSources" Class="
                EquivalentSources" ID="8">
73            <Component Name="Materials" Reference="Materials"
```

```
            />
          <Component Name="Models" Reference="Model"/>
        </Variable>
        <Variable Name="MyModel_1" Class="Model" ID="9">
          <Component Name="Parts" Type="String">
            <Size> 1</Size>
            <Value> BoxPart_1</Value>
          </Component>
        </Variable>
        <Variable Name="MyModel_2" Class="Model" ID="10">
          <Component Name="Parts" Type="String">
            <Size> 1</Size>
            <Value> BoxPart_2</Value>
          </Component>
        </Variable>
        <Variable Name="MyModel_3" Class="Model" ID="11">
          <Component Name="Parts" Type="String" Reference="
             Part">
            <Size> 1</Size>
            <Value> PlatePart</Value>
          </Component>
          <Component Name="Parameters"/>
        </Variable>
        <Variable Name="BoxPart_1" Class="Part" ID="12">
          <Component Name="CSGDefinition" Type="String">
            <Value> SolidBox_1</Value>
          </Component>
          <Component Name="BReps" Type="String">
            <Size> 1</Size>
            <Value> Box_1Mesh</Value>
          </Component>
        </Variable>
        <Variable Name="BoxPart_2" Class="Part" ID="13">
          <Component Name="CSGDefinition" Type="String">
            <Value> SolidBox_2</Value>
          </Component>
          <Component Name="BReps" Type="String">
            <Size> 1</Size>
            <Value> Box_2Mesh</Value>
```

```
111           </Component>
112         </Variable>
113         <Variable Name="PlatePart" Class="Part" ID="14">
114           <Component Name="CSGDefinition" Type="String">
115             <Value> RectangularPlate</Value>
116           </Component>
117           <Component Name="BReps" Type="String" Reference="
                BRep">
118             <Size> 1</Size>
119             <Value> PlateMesh</Value>
120           </Component>
121         </Variable>
122       </Folder>
123       <Folder Name="Geometry" ID="2">
124         <Variable Name="Operator" Class="Operator" ID="15">
125           <Component Name="CSGElements" Reference="
                CSGElement"/>
126         </Variable>
127         <Variable Name="Points" Class="Points" ID="16">
128           <Structure>ListOfTuples</Structure>
129           <Component Name="Coordinates" Units="m" Type="
                double">
130             <Size> 3</Size>
131           </Component>
132           <Sizes>20</Sizes>
133         </Variable>
134         <Variable Name="BRepApplicability" Class="
              BRepApplicability" ID="17">
135           <Component Name="FrequencyRange" Units="Hz" Type=
                "double">
136             <Size> 2 </Size>
137           </Component>
138           <Component Name="Name" Type="string"/>
139           <Component Name="ReferenceValue" Type="double"/>
140           <Component Name="Range" Type="double">
141             <Size> 2 </Size>
142           </Component>
143           <Component Name="ModellingMethods" Type="strings"
                />
```

```
144          </Variable>
145          <Variable Name="VolumeMesh" Class="VolumeMesh" ID="
                18">
146            <Component Name="Roots" Type="integer"
                  Association="RootBricks"/>
147            <Component Name="MeshProperties" Type="integer"
                  Association="VolumeMeshTypes"/>
148            <Component Name="BRepApplicability" Association="
                  BRepApplicability"/>
149          </Variable>
150          <Variable Name="SurfaceMesh" Class="SurfaceMesh" ID
                ="19">
151            <Component Name="Roots" Type="integer"
                  Association="RootFaces"/>
152            <Component Name="MeshProperties" Type="integer"
                  Association="SurfaceMeshTypes"/>
153            <Component Name="BRepApplicability" Association="
                  BRepApplicability"/>
154          </Variable>
155          <Variable Name="CurveMesh" Class="CurveMesh" ID="20
                ">
156            <Component Name="Roots" Type="integer"
                  Association="RootEdge"/>
157            <Component Name="MeshProperties" Type="integer"
                  Association="CurveMeshTypes"/>
158            <Component Name="BRepApplicability" Association="
                  BRepApplicability"/>
159          </Variable>
160          <Variable Name="RootBricks" Class="RootBricks" ID="
                21">
161            <Component Name="BrickIndices" Type="integer"
                  Association="Bricks"/>
162            <Component Name="Arrangements" Type="integer"
                  Association="Arrangements"/>
163          </Variable>
164          <Variable Name="RootFaces" Class="RootFaces" ID="22
                ">
165            <Component Name="FaceIndices" Type="integer"
                  Association="Faces"/>
```

```
166           <Component Name="Arrangements" Type="integer"
                 Association="Arrangements"/>
167         </Variable>
168         <Variable Name="RootEdges" Class="RootEdges" ID="23
               ">
169           <Component Name="EdgeIndices" Type="integer"
                 Association="Faces"/>
170           <Component Name="Arrangements" Type="integer"
                 Association="Arrangements"/>
171         </Variable>
172         <Variable Name="CurveMeshTypes" Class="
               CurveMeshTypes" ID="24">
173           <Structure>ListOfTuples</Structure>
174           <Component Name="MeshOrder" Type="string"/>
175           <Component Name="MeshType" Type="string"/>
176         </Variable>
177         <Variable Name="SurfaceMeshTypes" Class="
               SurfaceMeshTypes" ID="25">
178           <Component Name="MeshOrder" Type="string"/>
179           <Component Name="MeshType" Type="string"/>
180           <Component Name="MinNoVertices" Type="integer"/>
181           <Component Name="MaxNoVertice" Type="integer"/>
182         </Variable>
183         <Variable Name="VolumeMeshTypes" Class="
               VolumeMeshTypes" ID="26">
184           <Component Name="MeshOrder" Type="string"/>
185           <Component Name="MeshType" Type="string"/>
186           <Component Name="MinNoVertices" Type="integer"/>
187           <Component Name="MaxNoVertice" Type="integer"/>
188         </Variable>
189         <Variable Name="ReferenceSystem" Class="
               ReferenceSystem" ID="27">
190           <Structure>ListOfTuples</Structure>
191           <Component Name="Origin" Type="float">
192             <Size> 3 </Size>
193           </Component>
194           <Component Name="Orientation" Type="float">
195             <Size> 3 </Size>
196           </Component>
```

```
197           <Component Name="AssociatedElement" Reference="
                 Element"/>
198         </Variable>
199       </Folder>
200       <Folder Name="Topology" ID="3">
201         <Variable Name="Bricks" Class="Bricks" ID="28">
202           <Structure>ListOfTuples</Structure>
203           <Component Name="Face" Type="integer" Association
                 ="Faces">
204             <Value> 0 6</Value>
205           </Component>
206           <Sizes>0</Sizes>
207           <Component Name="Volume" Type="integer"
                 Association="Volumes"/>
208         </Variable>
209         <Variable Name="Faces" Class="Faces" ID="29">
210           <Structure>ListOfTuples</Structure>
211           <Component Name="AdjacentSheets" Type="integer"
                 Association="Sheets">
212             <Value> 1 3 5 7 9 11 13 15 17 19 21 23 25 27</
                    Value>
213             <Size> 1 </Size>
214           </Component>
215           <Sizes>14</Sizes>
216           <Component Name="Surface" Type="integer"
                 Association="Surface"/>
217         </Variable>
218         <Variable Name="Edges" Class="Edges" ID="30">
219           <Structure>ListOfTuples</Structure>
220           <Component Name="Face" Type="integer" Association
                 ="Faces"/>
221           <Sizes>28</Sizes>
222           <Component Name="Sibling" Type="integer"
                 Association="Siblings"/>
223           <Component Name="EndVertices" Type="integer"
                 Association="Vertices">
224             <Size> 2 </Size>
225           </Component>
226           <Component Name="Curve" Type="integer"
```

```
            Association="Curve"/>
227       </Variable>
228       <Variable Name="Vertices" Class="Vertices" ID="31">
229         <Structure>ListOfTuples</Structure>
230         <Component Name="Point" Type="integer"
              Association="Points"/>
231         <Sizes>20</Sizes>
232         <Component Name="Sibling" Type="integer"
              Association="Sibling"/>
233       </Variable>
234       <Variable Name="Sheets" Class="Sheets" ID="32">
235         <Structure>ListOfTuples</Structure>
236         <Component Name="Brick" Type="integer"
              Association="Bricks"/>
237         <Sizes>28</Sizes>
238         <Component Name="Face" Type="integer" Association
              ="Faces"/>
239         <Component Name="Sibling" Type="integer"
              Association="Siblings">
240           <Value> 0 0 1 2 3 4 12 12 13 14 15 16 24 24</
                Value>
241         </Component>
242       </Variable>
243       <Variable Name="Siblings" Class="Siblings" ID="33">
244         <Structure>ListOfTuples</Structure>
245         <Component Name="Edge" Type="integer" Association
              ="Edges"/>
246         <Sizes>28</Sizes>
247         <Component Name="Sheet" Type="integer"
              Association="Sheets"/>
248         <Component Name="IncidentSiblings" Type="integer"
               Association="Siblings">
249           <Size> 2 </Size>
250         </Component>
251         <Component Name="FaceTwin" Type="integer"
              Association="Siblings"/>
252         <Component Name="Vertex" Type="integer"
              Association="Vertices"/>
253         <Component Name="Sibling" Type="integer"
```

```
            Association="Sibling"/>
254         <Component Name="Cousin" Type="integer"
            Association="Cousin"/>
255       </Variable>
256     </Folder>
257     <Folder Name="ParameterSpace" ID="4">
258       <Variable Name="Parameter" Class="Parameter" ID="34
          ">
259         <Component Name="Name" Units="none" Type="string"
            />
260         <Component Name="Expression" Units="none" Type="
            string"/>
261         <Component Name="MinValue" Units="none" Type="
            double"/>
262         <Component Name="MaxValue" Units="none" Type="
            double"/>
263       </Variable>
264     </Folder>
265     <Folder Name="Materials" ID="5">
266       <Variable Name="Aluminium" Class="ConductingSurface
          " ID="35">
267         <Component Name="ElectricalConductivity" Type="
            double" Units="S/m">
268           <Value> 3.816e7</Value>
269         </Component>
270         <Component Name="MagneticPermeability" Type="
            double" Units="H/m">
271           <Value> 1</Value>
272         </Component>
273         <Component Name="Frequency" Type="double" Units="
            Hz">
274           <Value> 12e9</Value>
275         </Component>
276       </Variable>
277       <Variable Name="HomogeneousIsotropicMedium" Class="
          HomogeneousIsotropicMedium" ID="36">
278         <Component Name="DieliectricPermittivity" Units="
            F/m" Type="double"/>
279         <Component Name="MagneticConductivity" Units="Ohm
```

```
            /m" Type="Double"/>
280       </Variable>
281       <Variable Name="Boundaries" Class="Boundaries" ID="
              37">
282         <Component Name="IncidenceAngleTheta" Units="rad"
                Type="Double"/>
283         <Component Name="IncidenceAnglePhi" Units="rad"
              Type="Double"/>
284         <Component Name="Frequency" Units="Hz" Type="
              Double"/>
285       </Variable>
286       <Variable Name="ReflectionCoefficient" Class="
              ReflectionCoefficient" ID="38">
287         <Component Name="ReflectionCoefficient" Units="
              none" Type="Complex"/>
288       </Variable>
289       <Variable Name="SurfaceImpedance" Class="
              SurfaceImpedance" ID="39">
290         <Component Name="SurfaceImpedance" Units="Ohm"
              Type="Complex"/>
291       </Variable>
292       <Variable Name="SurfaceAdmittance" Class="
              SurfaceAdmittance" ID="40">
293         <Component Name="SurfaceAdmittance" Units="S"
              Type="Complex"/>
294       </Variable>
295       <Variable Name="Media" Class="Media" ID="41">
296         <Component Name="IncidenceAngleTheta" Units="rad"
                Type="Double"/>
297         <Component Name="IncidenceAnglePhi" Units="rad"
              Type="Double"/>
298         <Component Name="Frequency" Units="Hz" Type="
              Double"/>
299       </Variable>
300       <Variable Name="ReflectionAndTransmissionMatrix"
              Class="ReflectionAndTransmissionMatrix" ID="42"
              >
301         <Component Name="ReflectionCoefficient" Units="
              none" Type="Complex"/>
```

```
302           <Component Name="TransmissionCoefficient" Units="
                  none" Type="Complex"/>
303         </Variable>
304         <Variable Name="ScatteringMatrix" Class="
                ScatteringMatrix" ID="43">
305           <Component Name="ScatteringElements" Units="none"
                   Type="Complex"/>
306         </Variable>
307         <Variable Name="ImpedanceMatrix" Class="
                ImpedanceMatrix" ID="44">
308           <Component Name="ImpedanceElements" Units="Ohm"
                  Type="Complex"/>
309         </Variable>
310         <Variable Name="AdmittanceMatrix" Class="
                AdmittanceMatrix" ID="45">
311           <Component Name="AdmittanceElements" Units="S"
                  Type="Complex"/>
312         </Variable>
313       </Folder>
314     </Folder>
315   </Declarations>
316   <!-- === Data section === -->
317   <Data>
318     <Variable Name="Points" RefID="15">
319       <Component Name="Coordinates" Type="double">
320         <Value>3.0 1.0 3.0 0.0 1.0 3.0 0.0 0.0 3.0 3.0 0.0
              3.0 3.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 3.0 0.0
               0.0 3.0 1.0 6.0 0.0 1.0 6.0 0.0 0.0 6.0 3.0
              0.0 6.0 3.0 1.0 3.0 0.0 1.0 3.0 0.0 0.0 3.0 3.0
               0.0 3.0 3.0 1.0 0.0 5.0 1.0 0.0 5.0 3.0 0.0
              3.0 3.0 0.0</Value>
321       </Component>
322     </Variable>
323     <Variable Name="Edges" RefID="29">
324       <Component Name="Face" Type="integer">
325         <Value>0 0 0 0 1 1 1 2 2 3 3 4 6 6 6 6 7 7 7 8 8 9
              9 10 12 12 12 12</Value>
326       </Component>
327       <Component Name="Sibling" Type="integer">
```

```
328         <Value>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
                18 19 20 21 22 23 24 25 26 27</Value>
329       </Component>
330       <Component Name="EndVertices" Type="integer">
331         <Value>0 1 1 2 2 3 3 0 4 5 5 1 0 4 5 6 6 2 6 7 7 3
                7 4 8 9 9 10 10 11 11 8 0 1 1 9 8 0 1 2 2 10 2
                3 3 11 3 0 4 17 17 18 18 19 19 4</Value>
332       </Component>
333     </Variable>
334     <Variable Name="Vertices" RefID="30">
335       <Component Name="Point" Type="integer">
336         <Value>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
                18 19</Value>
337       </Component>
338       <Component Name="Sibling" Type="integer">
339         <Value>0 0 1 2 4 4 7 9 8 8 9 10 12 12 15 17 16 16
                17 18</Value>
340       </Component>
341     </Variable>
342     <Variable Name="Sheets" RefID="31">
343       <Component Name="Brick" Type="integer">
344         <Value>0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
                1 1 nan nan nan nan</Value>
345       </Component>
346       <Component Name="Face" Type="integer">
347         <Value>0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10
                10 11 11 12 12 13 13</Value>
348       </Component>
349     </Variable>
350     <Variable Name="Siblings" RefID="32">
351       <Component Name="Edge" Type="integer">
352         <Value>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
                18 19 20 21 22 23 24 25 26 27</Value>
353       </Component>
354       <Component Name="Sheet" Type="integer">
355         <Value>0.0 2.0 0.0 4.0 0.0 6.0 0.0 8.0 2.0 10.0 2.0
                 4.0 2.0 8.0 4.0 10.0 4.0 6.0 6.0 10.0 6.0 8.0
                8.0 10.0 12.0 14.0 12.0 16.0 12.0 18.0 12.0
                20.0 14.0 22.0 14.0 16.0 14.0 20.0 16.0 22.0
```

```
            16.0 18.0 18.0 22.0 18.0 20.0 20.0 22.0 24.0
            26.0 24.0 26.0 24.0 26.0 24.0 26.0</Value>
      </Component>
      <Component Name="IncidentSiblings" Type="integer">
        <Value>3 5 0 8 1 10 2 6 6 7 4 1 0 11 5 9 7 2 8 11 9
             3 10 4 11 1 8 16 9 18 10 2 2 3 0 9 8 19 1 17 3
             10 16 19 17 11 18 0 19 17 4 18 17 19 18 4 1 6
            2 5 3 8 0 10 5 11 0 7 4 3 8 4 1 9 10 7 2 11 6 9
             9 2 10 1 11 16 8 18 1 19 8 3 0 11 16 0 9 17 18
             3 10 19 2 17 17 19 18 4 19 17 4 18</Value>
      </Component>
      <Component Name="FaceTwin" Type="integer">
        <Value>16 19 21 23 nan nan nan nan nan nan nan nan
            nan nan nan nan 0 nan nan 1 nan 2 nan 3 nan nan
             nan nan</Value>
      </Component>
    </Variable>
  </Data>
  <!-- === Application Data section === -->
  <ApplicationData>
</ApplicationData>
</EDIFile>
```

9.2 Other examples

A number of other examples of increasing complexity and interest were created. The general method followed is the one outlined in the previous section and shown in the flow chart in Figure 9.4.

It is to note that while the previous and other simple examples were generated with the program itself, the larger ones presented in this section were generated with a commercial CAD (Rhinoceros) and imported using the Polygon File Format or Stanford Triangle Format (.ply) data format, which provides a basic but rather effective way to transfer the faceted geometries used for the testing.

The structures considered for a further discussion in this work are:

- a reflector antenna;

- a simplified satellite;
- the Emerald Satellite.

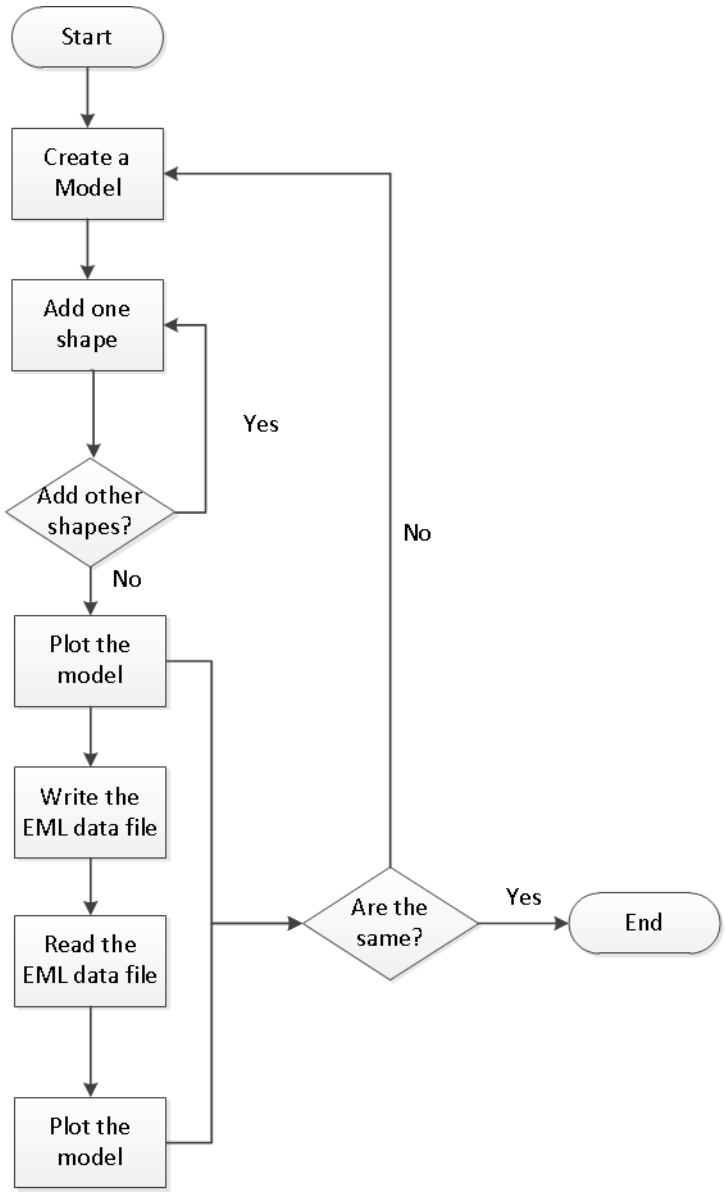


Figure 9.4: General method, flow chart.

9.2.1 Reflector Antenna

The reflector antenna considered is constituted by 45 different shapes merged together: 8 pyramids, all obtained by one single shape rotated and translated as needed, 4 parallelepiped, obtained as the previous shapes, one rhomboid as feeder and 32 triangles forming the paraboloid. Once again the shapes have different dimensionalities with the following topological consequences.

The CAD version can be seen in Figure 9.5, while the plot corresponding to the EML file is shown in Figure 9.6.
It is to note that the vertices of the pyramids are in common with the rhomboid in the upper ones and with some vertices of the reflector for the others and these points are correctly individuated and merged by the program.

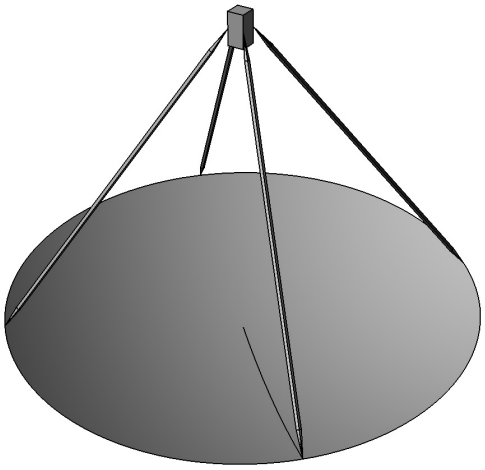


Figure 9.5: Reflector antenna, CAD model.

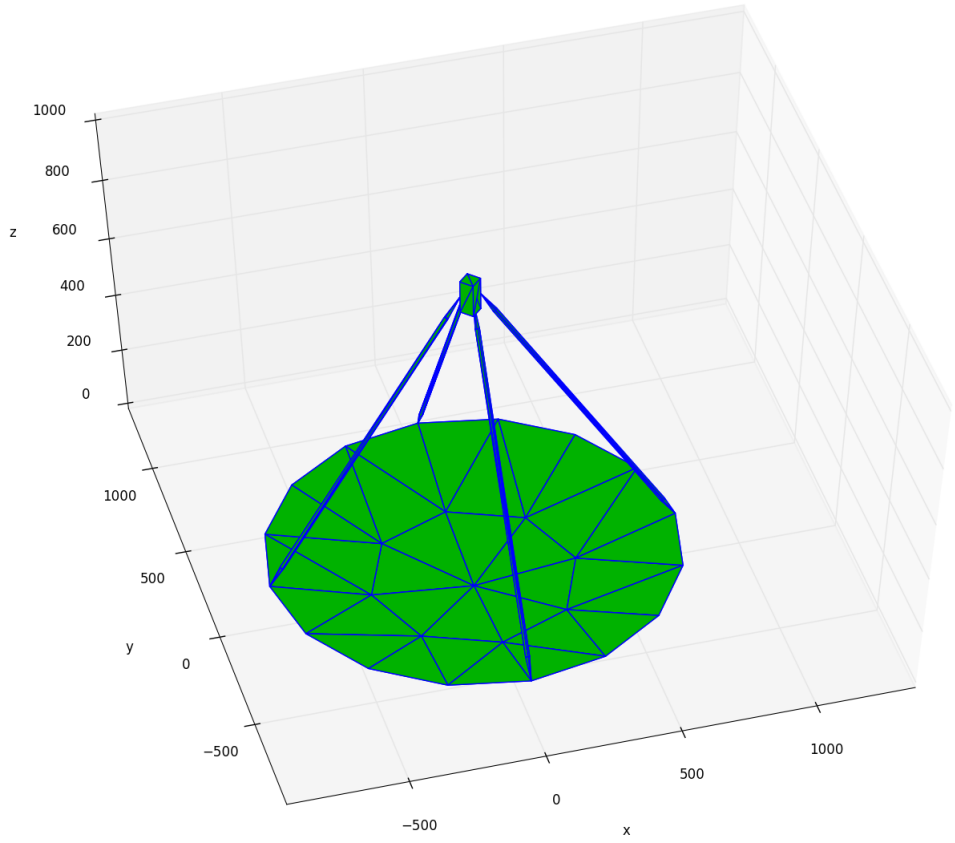


Figure 9.6: Reflector antenna plotted model.

9.2.2 Simplified Satellite

The simplified satellite considered had been obtained with the CAD software and it is shown in Figure 9.7.
This is actually the more complex example as the number of the involved shapes are almost one thousands.

In fact, the structure of this satellite is made of ten main parts: the solar panels, the body, three reflectors of the sides of the body, two other antennas on the top and two little horns on the superior edge of the body and an array on the back.

It is to highlight that the reflectors on the side are a completely different shape compared to the antennas on the top that actually are scaled version on the antenna described in section 8.2.1.

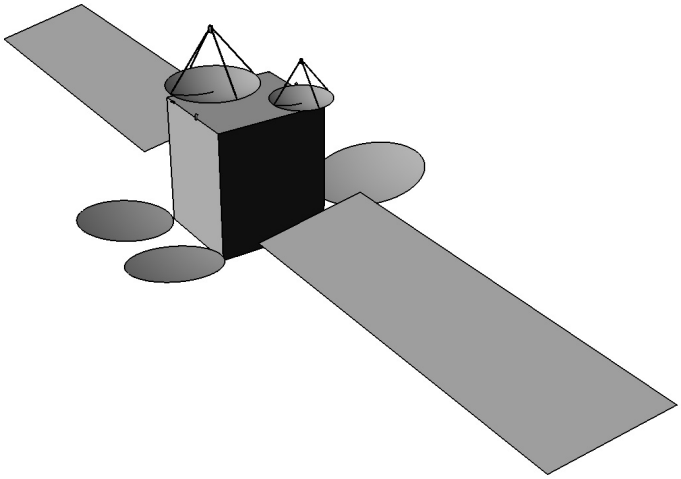
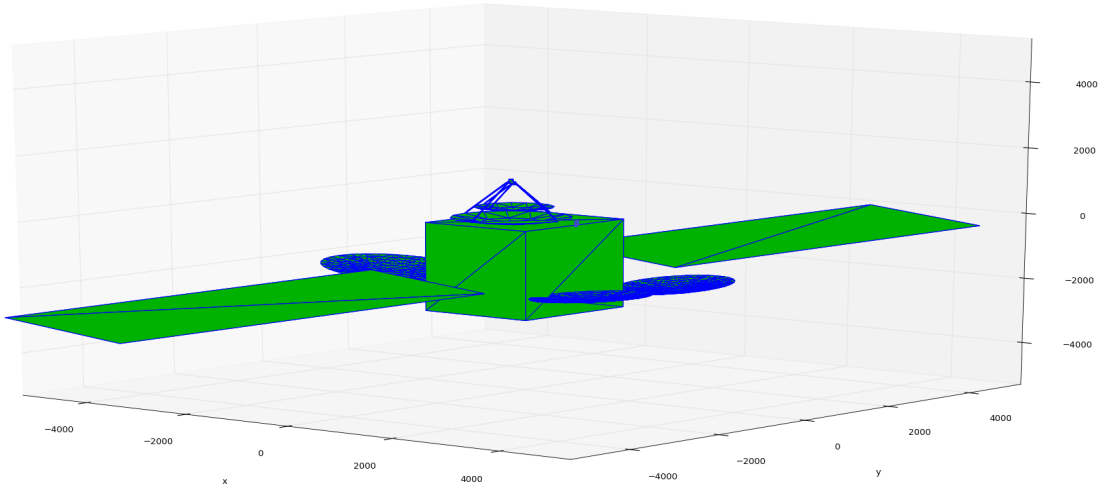


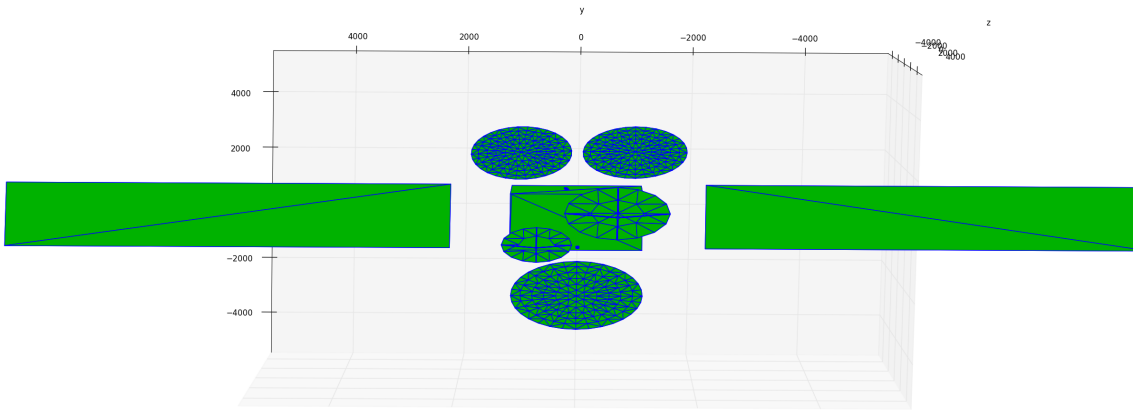
Figure 9.7: Simplified Satellite, CAD model.

The time required to compute all the matrices of the topological structure is about one hour and this is due to the fact that the tool has not been optimised but the only time that matter is the one required to actually write or read the data file. This time is about $25s$ in this case which is more than acceptable.

The obtained plots are shown in Figure 9.8 and some details are presented in Figure 9.10.

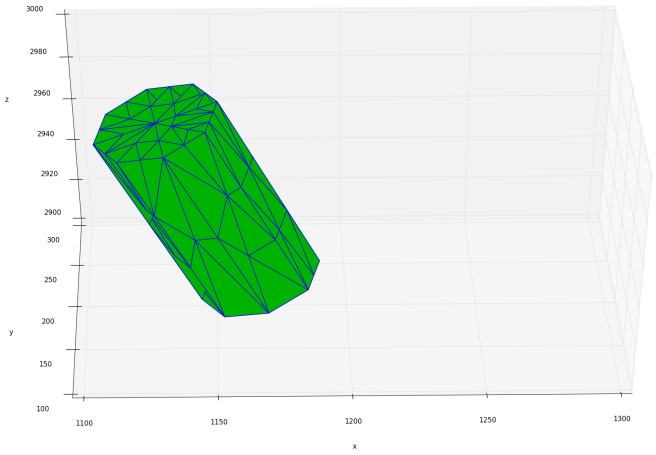


(a) *Lateral view.*

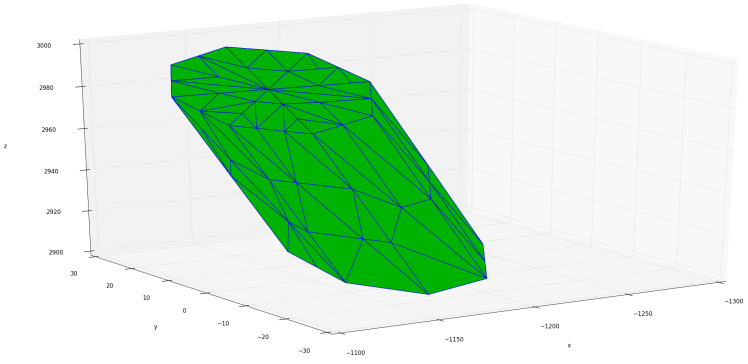


(b) *Top view.*

Figure 9.8: Simplified Satellite.

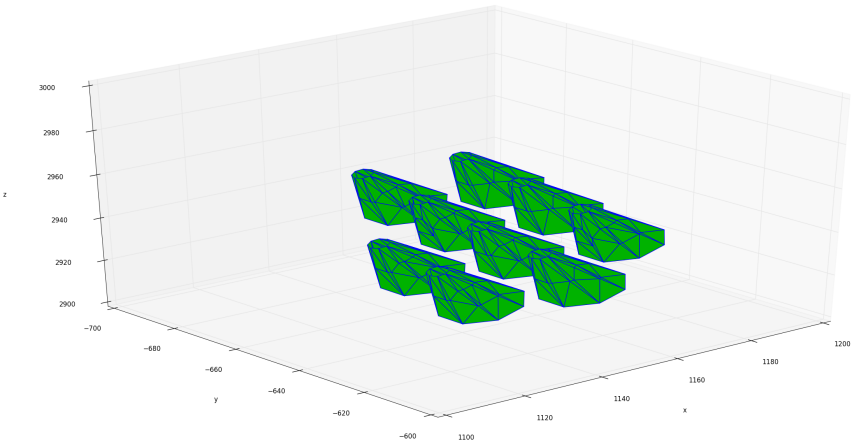


(a) *Horn A.*

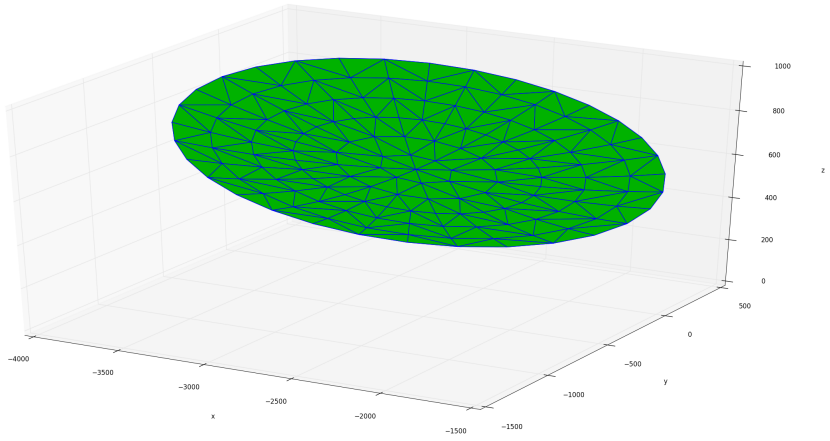


(b) *Horn B.*

Figure 9.9: Simplified Satellite details.



(a) *Array*.



(b) *Reflector*.

Figure 9.10: Other details of the Simplified Satellite.

9.2.3 Emerald Satellite

A model of the Emerald Satellite has been considered (Figure 9.11).
Using this, an entire work of CAD preconditioning had been applied and the complete analysis and description can be found in [28].

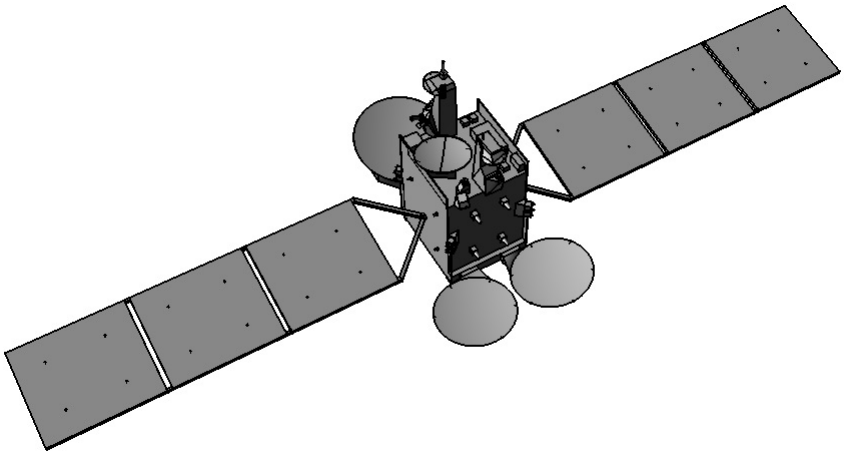


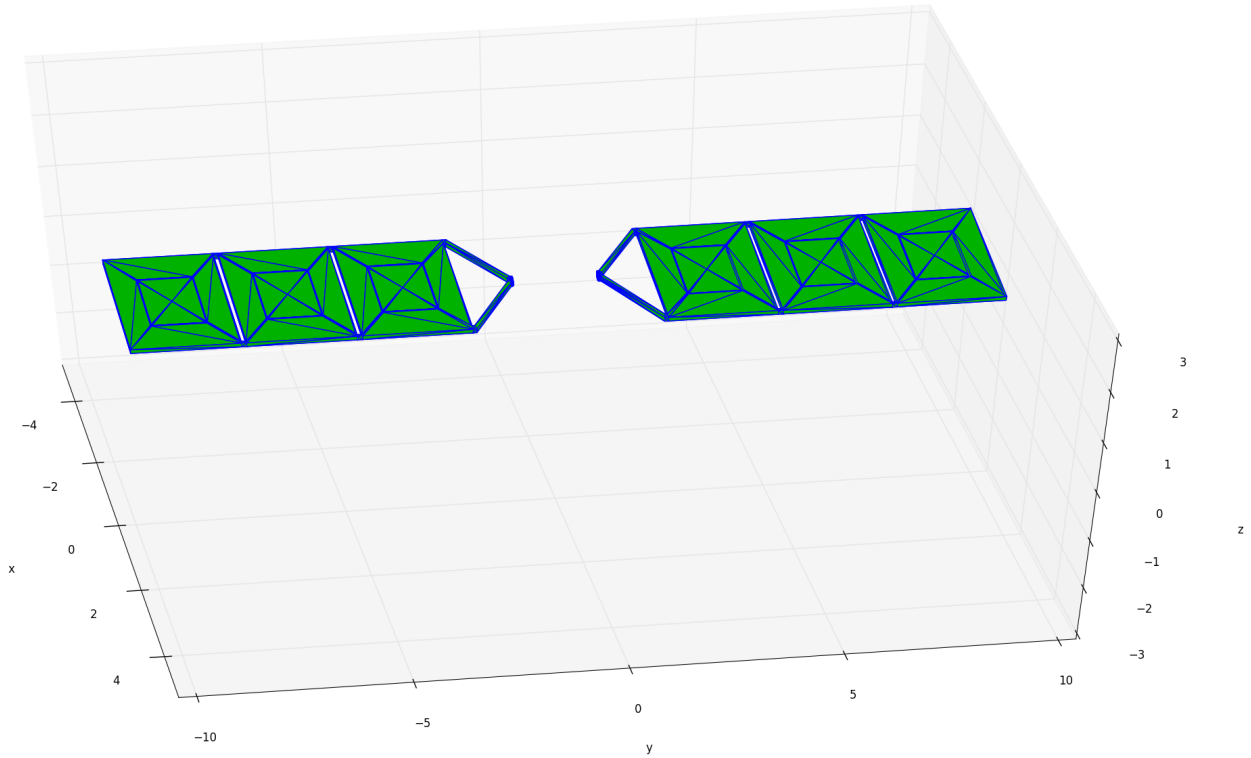
Figure 9.11: Emerald Satellite, CAD model.

The satellite then, had been meshed and divided into two parts corresponding to the solar panels and the body with the antennas and the equipment.
For the purpose of this work, each solar panel has been considered as single shape and so the other portion which includes the body.

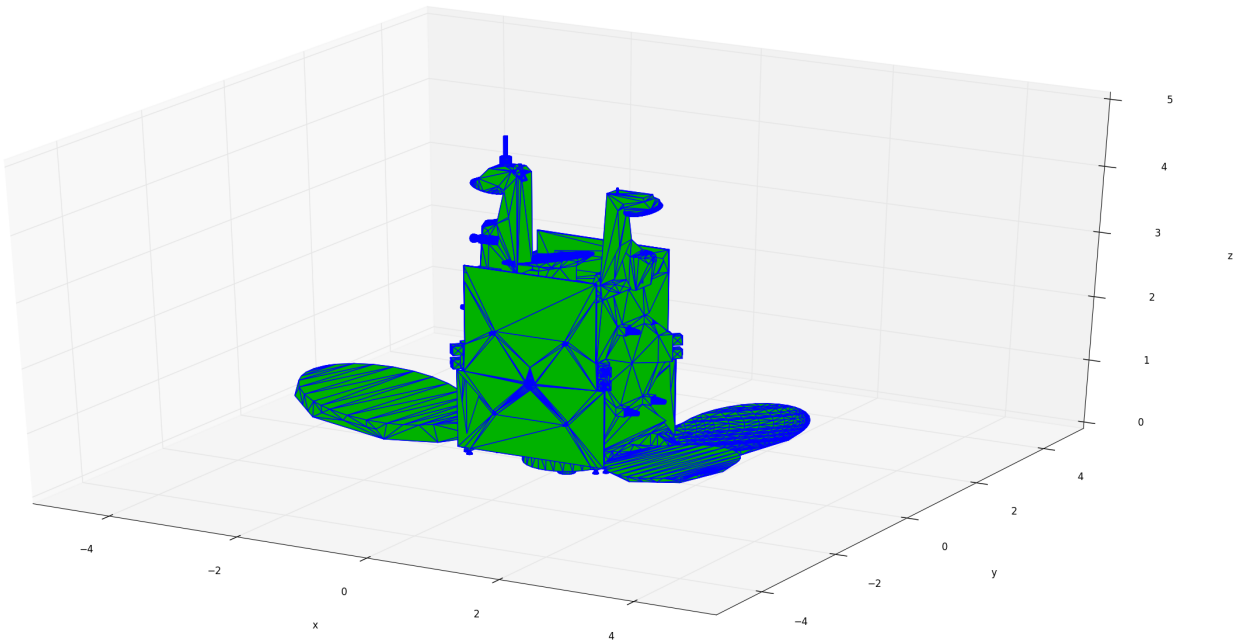
About the first part, one panel had been generated and added to the model, then the other one had been added and the two corresponding topological structure had been concatenated since, obviously, they do not share any points. The resulting structure have 2752 points and 5616 faces.

The body and the equipment, constituting one single shape, have a mesh made of 5887 points and 11778 triangles.

It is clear that the computational time required to generate all the matrices, operate and concatenate on them is much longer than the previous examples but, again, the time that really matter is the actual writing/reading time that is of about 60 s at most, for the body of this satellite.



(a) *Solar Panels.*



(b) *Body.*

Figure 9.12: Emerald Satellite.

Chapter 10

Conclusions

The design of antennas, especially space antennas, is a rather complex process that involves continuous adjustments and refinements until the desired result is reached. This implies an ongoing exchange of information between different tools and specialists from various discipline and most of the time the data have to be translated or adjusted to the different languages, standards or habits. This process not only consume a lot of time but also impede to reach better results that could be achieved more easily if there were a common language shared from all the design tools and used by all the specialists.

To reach this objective, that had been felt more and more from the community during the last decade, the Electromagnetic Data Exchange (EDX) Working Group was founded and it is composed by the Electromagnetic and Space Division of the European Space Agency, the Antenna Centre of Excellence and the European Antenna Modelling Library team.

The outcome is the Electromagnetic Data Exchange (EDX) language. It is formed by three main elements: a neutral XML-based Electromagnetic Markup Language (EML), with a simple grammar that is used for the data files, a set of Electromagnetic Data Dictionaries (EDDs) establish the lexicon of the exchange language and a software library, the Electromagnetic Data Interface (EDI), that simplifies the access to data from C++, Fortran and Matlab programs.

The Electromagnetic Data Dictionary initially identified concern the fields,

the induced currents on various geometries, the Green's function for layered structures, circuit parameters, modal expansion and the geometry.

A number of results have been achieved (Chapter 8) both theoretically and practically.
Concerning the first kind, the most important is the implementation of the geometry data model called Structure Data Dictionary. The elements that had to appear on this EDD were already established, but the actual data structure, how all the information have to be related and organised and a number of other details have been developed and discussed in this work (see Chapter 7). It is relevant to note that, when considering a complex structure interesting for electromagnetic purposes, e.g. a satellite, the information needed for the design and analysis purposes, includes a number of information that are not strictly related to the geometry of the object at first sight, such as material properties. This is one of the main reason why the Structure Data Dictionary (SDD) has a much higher level of complexity if comparing it to the Field Data Dictionary previously developed by the same Working Group. The SDD consist in four main folders contained in a common frame (Root Folder) that include some specification shared by all the folders. These last are the Objects Folder, the Geometry, the Topology, the Materials and the Parameter Space one. Each of those includes a number of classed related and connected to each other in a way to furnish all the information required and a reasonable access to them.

The most remarkable practical result of this work is the implementation of a Python-based prototype tool, a minimalistic CAD, to manage the physical structure data model with particular attention to the geometrical and topological information. This tool in composed by three main logical segments: the first one is the EDX I/O back end, that contains a complete sets of functions to fully access the data files. The second part is the Data Domain Model File (DMF) loader, which is constituted by an implementation of a lexical (LEX) and syntactical (YACC) analyser able to read and check the formal definition of the EDX language and create a dictionary containing all the information on the data model needed to fill up the EML file.

The third segment is the Geo Modeller that includes a library of functions to assemble the geometrical and topological information and encode them in another dictionary containing the actual values that has to be written or that had been read on the EML data file.
The coordination and interactions between the different modules are managed by a single main script that inherently has many degree of freedom to give the user the possibility to employ it fully. Each portion of the program can also be used independently as stand alone tool since they have been developed at different times and integrated only in the last stages of the study.

This prototype has been exploited to generate in an easy and effective way, some complex and complete examples of the Structure Data Dictionary (see Chapter 8). In particular, starting from some basic examples, two type of reflector antennas, an array, a simplified satellite and the Emerald Satellite have been considered.
The general method can be summarised as follow: an empty model is created, then one or more shapes are generated and their topological structure is merged, concatenated or unified depending on the case and then added to the model. Eventually the model is plotted and the corresponding EDX data file is written. Afterwards the data file is read, the model is rebuilt and is it plotted, if the initial shape and this last one displayed are the same we are reasonably sure that the data structure is complete and correct and the corresponding EML data file is valid.

While the Structure Data Dictionary has reach its ultimate version and has been formalised, some future improvement are foreseen for the prototype tools. Particular effort will be put on the optimisation of the code and on the improvements of the routine for generate the other non-topological information.

Appendix A

XML example file

The following example had been created using the Field Data Dictionary and in particular the Far Field class there defined.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <EDIFile  xmlns="http://www.edi-forum.org"
3            xmlns:xsi="http://www.w3.org/2001/XMLSchema-
                 instance"
4            xsi:schemaLocation="http://www.edi-forum.org edi.
                 xsd">
5  <!-- === Header section === -->
6  <Header>
7    <Stamps>
8      <Version>EDI Version 1.00.00</Version>
9      <Format>XML</Format>
10     <DateTime>2006-12-01T12:40:28Z</DateTime>
11   </Stamps>
12   <Origin>
13     <Tool><Name></Name><Version></Version></Tool>
14     <Project></Project>
15     <User>
16       <Name></Name>
17       <Affiliation></Affiliation>
18     </User>
19   </Origin>
```

```
20    <UserText></UserText>
21  </Header>
22  <!-- === Declarations section === -->
23  <Declarations>
24    <Folder Name="EDI_FF_TestFile_2.xml" ID="0">
25      <Variable Name="Horn_Directivity" Class="
            Field:Directivity" ID="10">
26        <Sizes> 2 3 2 2</Sizes>
27        <Domain Reference="Horn_ProjectionComponents"/>
28        <Domain Reference="Horn_Phi"/>
29        <Domain Reference="Horn_Theta"/>
30        <Domain Reference="Horn_Frequency"/>
31        <Component Type="double"/>
32      </Variable>
33      <Variable Name="Horn_Field" Class="Field:Far" ID="1">
34        <Attribute Name="SpaceTypeAxis">Space</Attribute>
35        <Attribute Name="TimeDependency">+j\omegat</Attribute
              >
36        <Attribute Name="TimeTypeAxis">Frequency</Attribute>
37        <Sizes></Sizes>
38        <Component Reference="Horn_Frequency"/>
39        <Component Reference="Horn_ScanRange_2D"/>
40        <Component Reference="Horn_ProjectionComponents"/>
41        <Component Reference="Horn_PowerNormalisation"/>
42        <Component Reference="Horn_RelativeGainOffset"/>
43        <Component Reference="Horn_PhaseReference"/>
44        <Component Reference="Horn_Directivity"/>
45      </Variable>
46      <Variable Name="Horn_Frequency" Class="Frequency" ID="2
            ">
47        <Sizes> 2</Sizes>
48        <Component Type="double">
49          <Value> 5 7</Value>
50        </Component>
51      </Variable>
52      <Variable Name="Horn_PhaseReference" Class="
            PhaseReferencePoint" ID="9">
53        <Sizes></Sizes>
54        <Component Name="x" Type="double">
```

```
55          <Value> 5</Value>
56        </Component>
57        <Component Name="y" Type="double">
58          <Value> 5</Value>
59        </Component>
60        <Component Name="z" Type="double">
61          <Value> 5</Value>
62        </Component>
63      </Variable>
64      <Variable Name="Horn_Phi" Class="Phi" ID="5">
65        <Sizes> 3</Sizes>
66        <Component Type="double">
67          <Value> 2 4 8</Value>
68        </Component>
69      </Variable>
70      <Variable Name="Horn_PowerNormalisation" Class="
           PowerReference" ID="7">
71        <Sizes></Sizes>
72        <Component Name="Radiated" Type="double">
73          <Value> -1</Value>
74        </Component>
75        <Component Name="Accepted" Type="double">
76          <Value> -1</Value>
77        </Component>
78        <Component Name="MatchedLine" Type="double">
79          <Value> -1</Value>
80        </Component>
81        <Component Name="Available" Type="double">
82          <Value> -1</Value>
83        </Component>
84      </Variable>
85      <Variable Name="Horn_ProjectionComponents" Class="
           ProjectionComponents:Spherical" ID="6">
86        <Sizes> 2</Sizes>
87        <Component Type="string">
88          <Value> "Theta" "Phi"</Value>
89        </Component>
90      </Variable>
```

```
    <Variable Name="Horn_RelativeGainOffset" Class="
        RelativeGainNormalisationOffset" ID="8">
      <Sizes></Sizes>
      <Component Type="double">
        <Value> 0</Value>
      </Component>
    </Variable>
    <Variable Name="Horn_ScanRange_2D" Class="
        ScanRange:ThetaPhi" ID="3">
      <Sizes></Sizes>
      <Component Reference="Horn_Theta"/>
      <Component Reference="Horn_Phi"/>
    </Variable>
    <Variable Name="Horn_Theta" Class="Theta" ID="4">
      <Sizes> 2</Sizes>
      <Component Type="double">
        <Value> 3 5</Value>
      </Component>
    </Variable>
  </Folder>
</Declarations>
<!-- === Data section === -->
<Data>
  <Variable Name="Horn_Directivity" RefID="10">
      <Component Type="double">
        <Value>
 1.1 1.2
 2.1 2.2
 21.1 21.2
 22.1 22.2
 31.1 31.2
 32.1 32.2
 -1.1 -1.2
 -2.1 -2.2
 -21.1 -21.2
 -22.1 -22.2
 -31.1 -31.2
 -32.1 -32.2
        </Value>
```

```
128         </Component>
129     </Variable>
130 </Data>
131 <!-- === Application Data section === -->
132 <ApplicationData>
133 </ApplicationData>
134 </EDIFile>
```

Appendix B

Synopsis of the EDX Data Dictionary Declaration Language

The full description and documentation can be found in [31].

```
  Data dictionary <dictionary name>
    [includes<dictionary name>*{,<dictionary name>}]
    +{ClassDeclaration}
  end

ClassDeclaration =
  class <class name>
  [ NewClassDeclaration |
    SubclassDeclaration |
    ClassImportDeclaration]
  [ ClassRulesDeclaration ]
  [ ClassMembersDeclaration ]
  end

NewClassDeclaration =
  ([ new ] | override)
  *{attribute <attribute name> : <value>*{,<value>}}
  *{domain <domain name> reference <class name>}
  [structure ( CartesianProduct | ListOfTuples )]
  [
```

```
    units <unit symbol>
    type <type name>
    size <number> *{ , <number> }
  |
    +{ component <component name>
      (
        units <unit symbol>
        type <type name>
        [ size <number>*{,<number>}]
        [ association<class name>]
      |
        reference <class name>
      )
    }
  ]

SubclassDeclaration =
  extends <class name>
  *{attribute<attribute name>values <value>*{,<value>}}
  *{domain<domain name> reference <class name>}
  *{component<component name>
    (
      units <unit symbol>
      type <type name>
      [ size <number> *{,<number>}]
      [association <class name>]
    |
      reference <class name>
    )
  }


ClassImportDeclaration =
  alias[<data dictionary name>::]<class name>*{:<class name
     >}

ClassRulesDeclaration =
  rules
  [structure
```

```
+{mandatory <component name>  *{ , <component name> } |
optional <component name>  *{ , <component name> } |
present <component name> with <component name> |
present <component name>
  when <component name> = <component value> |
optional <component name>
  when <component name> = <component value> |
presentAtLeastOne <component name> *{ , <component name>
   } |
presentOnlyOne <component name> *{ , <component name> }
}
end]
[integrity
 +{ component <component name> (
  allowed values: ( <value> *{ , <value> } |
  components names in <class name> )
  bounds ( [ <symbolicValue> , <symbolicValue> ] |
  ( <symbolicValue> , <symbolicValue> ) )
  *{,<symbolicExpression> } |
  relation <symbolicExpression>)
}
end]
[transformations
[permutation <integervalue> <integervalue>*{,<
   integervalue>}]
[ordering <ordering declaration>]
[sampling <sampling pattern>]
[slicing <slicing pattern>]
end]
[ defaults
[ variableName <name> ]
[ structure ( CartesianProduct | ListOfTuples ) ]
 end ]
[ style
[ numberFormat <formatstring> ; ]
[ complex ( Cartesian | Polar ) ]
 end ]
end
```

```
ClassMembersDeclaration =
  members
  +{ variable <variable name>
  *{ attribute <attribute name> : <value> *{ , <value> } }
  *{ domain <domain name> reference <class name> }
  [ structure ( CartesianProduct | ListOfTuples ) ]
  [
    units <unit symbol>
    type <type name>
    [ size <number> *{ , <number> } ]
    value <<list of values>>
  |
  +{ component <component name>
    (
    units <unit symbol>
    type <type name>
    [ size <number> *{ , <number> } ]
    (
    value<<list of values>>|association <variable name>
    )
    | reference <variable name> )
  }]
  end
  }
end
```

Bibliography

[1] Marco Sabbadini *Data File Format for EM Modelling*, XEA/098.94, rev.1, 1994.

[2] P. E. Frandsen, M. Ghilardi, M. Sabbadini et al. *Requirements for the Eletromagnetic Data Interface (EDI) - XML file format and FORTRAND API*, joint EAML/ACE ASI File Formats activity, October 2005.

[3] J-P. Martinaud, P. E. Frandsen, G. Vandenbosch *The Ace Activity on strandardized file formats for electromagnetic software*, Proc. EuCAP, 2006.

[4] P. E. Frandsen, M. Ghilardi, F. Mioc et al. *The Electromagnetic Data Exchange: much more than a common data format*, Proc. EuCAP, 2007.

[5] F. Silvestri, M. Ghilardi *Electromagnetic Data Interface. EDI v.0.5 User Manual*, Information Technology Link srl, Protocol: MN/2006/001 Rev. 1.6, October 2006.

[6] F. Silvestri, M. Ghilardi *Electromagnetic Data Interface, FORTRAN User Manual - EDI Level 0 and Level 1*, ITLink Srl, Livorno, 2007.

[7] F. Silvestri, M. Ghilardi *Electromagnetic Data Interface, FORTRAN User Manual - EDI Level 2*, ITLink Srl, Livorno, 2007.

[8] F. Silvestri, M. Ghilardi *Electromagnetic Data Interface, MATLAB User Manual, EDI Level 2*, ITLink Srl, Livorno, 2007.

[9] F. Silvestri, M. Ghilardi *Electromagnetic Data Interface, C++ User Manual, EDI Level 2*, ITLink Srl, Livorno, 2007.

[10] Antenna Centre of Excellence *Deliverable Field Data Dictionary*, Contract FP6-IST 026957, March 2006.

[11] Antenna Centre of Excellence *Deliverable Current and Meshes Data Dictionary*, Contract FP6-IST 026957, 2005.

[12] Marco Sabbadini *Introduction to the design of Antennas for Space Applications.* Estec working paper N. 1963, 2nd Edition, 2006

[13] F. Mioc, M. Sabbadini *EDX. Field Data Dictionary definition*, ESA Ref.:EWP-2344, Issue 1, October 2008.

[14] Marco Sabbadini *System Analysis and Requirements for an Electromagnetic Data Exchange Standard*, ESA Ref.: EWP-2300, Issue 1, October 2005.

[15] Antenna Centre of Excellence *Deliverable Geometry Dictionary*, Contract FP6-IST 026957, 2006.

[16] P. E. Frandsen, T. Sejeroe *EDX Structures and Mesh DD. Proposal for EML Class Design Draft*, S1585-002, September 2013.

[17] F. Mioc, P.E. Frandsen, M. Sabbadini *EDX format for mesh exchange. Mesh dictionary*, Ref: TR.128.01.12.SATI.C, Draft Release C, May 2013

[18] Jean-Paul Martinaud, Poul Erik Frandsen, Guy Vandenbosch *The ACE activity on standardized file formats for electromagnetic software* Proc. EuCAP, Nice, November 2006.

[19] P. E. Frandsen, M. Sabbadini *EDX. An introduction to the Electromagnetic Data Exchange language.* ESA Ref.: EWP-2345, Issue 3, February 2009.

[20] M. Sabbadini, P.E. Frandsen, M. Ghilardi, G.A.E. Vandenbosch *EDX - Companion Tools and other. 2nd year developments*, Proc. EuCAP, Barcellona, April 2010.

[21] T. Bray, J. Paoli, C. M. Sperberg-McQueen et al. *http://www.w3.org/TR/xml11/*, September 2006.

[22] R. Rew, G. Davis *NetCDF: An Interface for Scientific Data Access* Unidata Program Center, IEEE 10.1109/38.56302, July 1990.

[23] *http://www.hdfgroup.org/HDF5*

[24] Marco Sabbadini *Parametric modelling of antenna and platform geometries*, Technical Note, ESA Ref.: TEC-EEA/2004.245/MS November 2004.

[25] Michael J. Pratt *Technical Note. Introduction to ISO 10303 - the STEP Standard for Product Data Exchange*, National Institute of Standards and Technology, Manufacturing Systems Integration Division, Gaithersburg, MD 20899-8261, USA.

[26] D. A. Schenk and P. R. Wilson *Information Modeling: The EXPRESS Way*, Oxford University Press, New York, NY USA, 1994

[27] Lucia Scialacqua, *Task 2: Intermediate Review Data Package* Ref: TR.041.01.12.SATI.A, 2012.

[28] Lucia Scialacqua *EAML VI, Task 1: Intermediate review data package*, Ref: TR.127.01.13.SATI.A, May 2013.

[29] *http://www.antennasvce.org/Community.*

[30] IEEE145-1993, *IEEE standard definition of terms for antennas. Antenna Standards Committee of the IEEE Antennas and Propagation Society.* March 1993.

[31] Marco Sabbadini, *Electromagnetic Data Exchange. Electromagnetic Declaration Language.* ESA Ref.: EWP-23xx, Issue 1, version B, November 2008.

[32] A.Pasko, V.Adzhiev, A.Sourin, V.Savchenko, *Function Representation in Geometric Modeling: Concepts, Implementation and Applications.* The Visual Computer, Springer, 1995.

[33] Alexander Pasko, Valery Adzhiev. *Function-based shape modeling: mathematical framework and specialized language* , Automated Deduction in Geometry, Springer, 2004.

[34] V Adzhiev at al., *Hyperfun project: A framework for collaborative multidimensional f-rep modeling*, 1999.

[35] *www.python.org*

[36] *http://www.dabeaz.com/ply/*

[37] *http://lxml.de/tutorial.html*

[38] J. D. Hunter, *Matplotlib: A 2D graphics environment*, Computing In Science & Engineering, Volume 9, IEEE COMPUTER SOC, 2007

[39] *http://www.numpy.org/*

[40] *www.cs.mtu.edu*

[41] J. Hald, J. E. Hansen, F. Jensen, F. Holm Larsen. *Spherical Near-Field Antenna Measurements*, Volume 26 of IEE electromagnetic Waves Series. Peter Peregrinus Ltd., 1998.

[42] Marco Sabbadini, *IDEA: Integrated Development Environment for Antennas*, Selenia Spazio, 1987.