Università degli Studi di Padova

Facoltà di Ingegneria.

Corso di Laurea in Ingegneria delle Telecomunicazioni.

Formal language for data models and software library for antenna data exchange

Laureando

Callegaro Giovanni

Relatore

Prof. Galtarossa Andrea

Co-relatore

Ing. Sabbadini Marco

Anno Accademico 2014/2015

A mio padre.

E a tutti coloro che hanno reso possibile questa magnifica esperienza.

Abstract

Oggetto di questa tesi è un resoconto delle attività svolte durante uno stage durato 6 mesi presso la sezione antenne di ESA/ESTEC (Noordwijk, Paesi Bassi). Scopo dell'attività è stato quello di perfezionare un sistema di scambio per dati di tipo elettromagnetico. EDX (Electromagnetic Data eXchange) nasce dall'esigenza di scambiare dati tra software di elaborazione diversi basandosi su modelli ben definiti in un apposito linguaggio chiamato DDL. In particolare si è lavorato sul ridefinire tale linguaggio e sull'aggiornare lo strumento di scambio che consiste in una libreria software scritta in C/C++, con interfacce Python e Fortran, alle versioni più recenti di tale linguaggio e all'esigenza di ampliare la varietà di strutture dati che possono essere immagazzinate, con un occhio alla retrocompatibilità. Questa libreria software nota con il nome di EDI (Electromagnetic Data Interface) è usata attualmente da diversi software commerciali di simulazione presso partner ed università.

Abstract

The subject of this thesis is a review on the activities performed during a six-month internship in the ESA/ESTEC (Noordwijk, The Netherlands) antenna section. The goal of the stage was to improve a system for the electromagnetic data exchange. EDX (Electromagnetic Data eXchange) is the answer to the need of exchanging data among different elaboration software using well-defined models. Such models are defined with a certain language called DDL that has been improved as well. The tool for the actual exchange is a C/C++ software library with Python and Fortran front-ends. The library has been drastically improved to support the new version of the language, to increase the variety of data structures that can be stored and to mantain the retrocompatibility. This library is known as EDI (Electromagnetic Data Interface) and it is currently used in several commercial simulation software by partners and universities.

Contents

1 Introduction **1**
1.1 Motivations 1
1.2 Quick overview 2

2 Background **5**
2.1 Dictionaries 6
2.2 EML 10
2.3 The EDI library 13

3 Language **17**
3.1 Improvements in the DDL 18
3.2 Improvements to the existing Dictionaries 22

4 Library **27**
4.1 Improvements in EML 27
4.2 Improvements in the library 30
4.3 Retrocompatibility 36

5 Conclusions and future developments **39**

A BNF and Railroad diagrams for the DDL **41**

Bibliography **51**

Chapter 1

Introduction

1.1 Motivations

Propagation problems, regardless of their physical dimension, have never been easy to handle. Tipically, the only approach to face these problems is by running simulations. Simulation software are usually based on Maxwell's equations which are difficult because they are differential equations. Plus, the number of points in meshes must be quite high as the frequencies are high, for example in telecommunications problems. In addition to this, if a system is complicated enough there could be several disturbance elements that simulations must take into account such as mechanical issues or unconsidered reflective surfaces.

In satellite transmissions, for example, the problem is really complicated as temperature excursions, mechanical stress during launches and limited amount of power available make simulations really hard to be managed in one go. Therefore, they have to be splitted in parts, with each part solved by someone competent in his own matter. Sometimes people collaborating to the common goal are not in strict contact or don't know much about each other's work: they even may work for different companies, they generally use a different software and work on different physical models.

The software used in these fields of work are usually very specialized and they generally make use of propietary data formats, even if they are handling the same physical quantities such as fields, currents, geometry, material properties, ranges in frequency, meshes and so on.

The day when all the data will be collected together to check if they are consistent, for being approved and for defining the ultimate solution, a huge amount of translations among models has to be conducted.

In order to address this problem, a couple of decades ago ESA pushed

for creating a language for exchanging electromagnetic data, called EDX. The Electromagnetic Data Exchange 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.

1.2 Quick overview

The contribution on the EDX language added during the internship period was based on a strong support. Basically every aspect of the matter has ben discussed before but at the same time many aspects were not mature. This thesis does not add much theoretical value to the matter but is limited to be a short review on changes that have been done on the system during late 2014. The changes are mostly practical as the software behind the system has been completely renewed.

The EDX system is currently and rapidly evolving, especially in the last few years. That is why an interested reader should care to look at the most recent documentation. In any case, chapter 2 is a really short briefing on what has been done so far and the current usage situation of EDX.

Data models concerning the same data sets are grouped in dictionaries. We will later explain in detail what the dictionaries are. In section 2.1 we introduce the main idea and we show some examples of Data Dictionaries.

Tha data sets are stored on files using EML that file format adopted for the exchange. In section 2.2 we introduce EML and show how the variables are stored.

In section 2.3 we give a very quick explanation on how EDI works. EDI is the software library that permits the actual saving/loading of EML files.

After having introduced the starting situation, the rest of the chapters report the main scores of the trainee period. Chapter 3 is about the changes in the DDL, the language that describes the dictionaries. This language has been improved a lot. The output of the parser for dictionaries is now directed to the EDI library which can now validate the variables stored with a deeper look at the meaning of the data.

Chapter 4 describes the improvements in the software library that actually do the work of exchanging data, storing it in several configurations and make a validation based on the dictionaries specified through the language. The library has been rewritten from scratch but the API has been mantained quite similar for retrocompatibility purposes. We will see how the new elements in the DDL have been transposed to new methods for the library.

In the conclusive chapter 5, we will list what are the next goals for EDX to be fulfilled in the near future.

Chapter 2

Background

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

$$\textbf{EDX} = \textbf{EDD's} + \textbf{EML} + \textbf{EDI}$$

Every one of these three aspects is related to the others. The data exchange is basically done with the use of EML files but if they were raw data files as such, they would be just a mere series of numbers that without context have no meaning at all. The purpose of EDX is to not only provide the raw data but to put it in its context and provide not only the numerical data but its location, purposes and reason of existence: has the data been measured or simulated? If so, what were the measurement conditions or the simulation tool? And what were the other parameters in game? And more, can these data be portable to be read on our own software? Contextualize the data in such a manner is the main reason EDX was created. That is why just a raw file is not sufficient to our purposes.

The EDDs introduce to the system the meanings that the data can assume. They provide data types introducing differencies among frequencies, currents, distancies, angles and various objects in general. They instruct the system on how these quantities are related to each other introducing entities and objects connected both vertically and horizontally to each other until they form the entire picture. For example, consider about a basic waveguide. From an electromagnetic point of view it is not a simple object but it

consists of a certain material with a certain shape that must be described with its own topology. Along the faces of the waveguide a certain current is present and so on. The system in consideration have to be decomposed and described in every single part of it. At that point there is no place for raw data: numerically storing all these aspects together is out of question. The system must then recognize the different nature of each object and the role of the dictionaries is to provide this.

The library is the point of connection among dictionaries and EML. It writes and reads EML files providing to the user not only the raw data but the basic elements realizations of dictionaries data sets which are called *variables*.

2.1 Dictionaries

The Data Dictionaries define the meaning of the data and the conventions for the exchange. A Data Dictionary include exactly and in detail all the elements that shall appear in a data set.

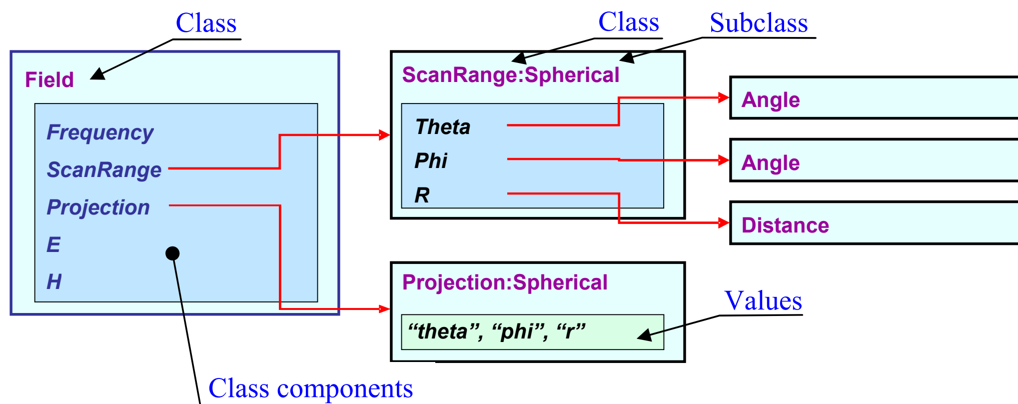


Figure 2.1: Example of decomposition for a Field object.

Dictionaries are composed of classes and every class represent a particular concept or object. It is very simple if we think of dictionaries classes as OOP classes. They are not concrete objects nor they have values but we can build instances of them, called *variables*. Classes are connected each other through the use of components or domains. For example, in figure 2.1 the *ScanRange:Spherical* class has three components: two angles and a

distance. The three of them are sufficient to describe a point of the scan range but the dimension of the scan range is another matter: of course three values won't describe a scan range. The *ScanRange* class should have a few implicit or explicit dimensions that will follow the components. The most straightforward way to describe such dimensiones is by the use of domains. Domains follow their mathematical concept. If we think of a class as a mathematical function it may have a domain and a codomain. In this case, the number of elements in the domains affect the number of the outcomes. According to the example, a projection of an electric Field has a different value according to its position and Frequency. Everyone of these values will belong to the projection component. This means that every component store a multidimensional matrix of values.

Another fundamental aspect of the dictionary is the inheritance and the abstraction of classes. Realizable or not, a class can have properties that other classes may inherit. Attributes, sizes, domains and components of a class will be present in each and every subclass of it. We can think of a generic measure that has been conducted. Its points of acquisition are described with some coordinates but in general we don't know what kind of coordinates we have (polar, cartesian...).

Anticipating what is a practical issue, we encounter the question of how to manage multidimensional quantities such as N-matrices in what components store. The answer is suggested in figure 2.2. We can think of the domains as independent quantities and the components, which describe the co-domain, as dependent quantities. The way of representing such a set of values is by the use of a vector that is formed by consecutive lines. This approach is really similar to the one that simple languages like C do when accessing to a multidimensional array on a volatile memory.

Summarizing, the classes of the dictionaries can present these kind of elements:

- Attributes, describing quality aspects
- Domains
- Sizes, about the class itself
- Components

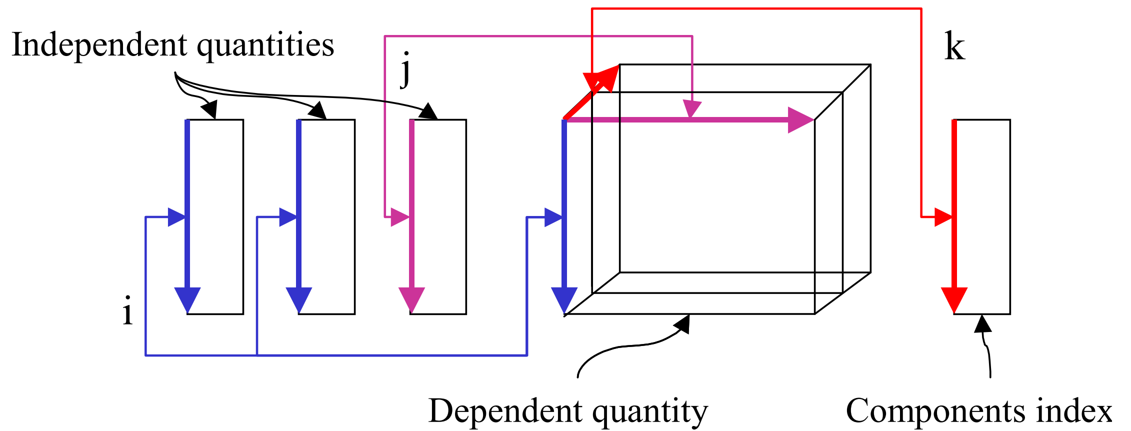


Figure 2.2: Dependent quantities

Six dictionaries have been initially identified to be included in the EDX project. Namely:

1. Fields (near, far and spherical wave expansion)
2. Induced currents on various geometries
3. Green's function for layered structures
4. Circuit parameters - [S], [Y] and [Z]
5. Modal expansions
6. Geometry (Structures).

As of today, the Field and the Currents Data Dictionaries are considered almost complete. For the moment, the Field DD is the only one being used by the partners while the Structure DD has been recently consolidated and is being experimentally applied in research activities.

An overview of the Structures (and Geometry) Data Dictionary can be found in figure 2.3. The dictionary is composed by 7 folders. Each one of them contain inherent classes and all together they can fully describe even complex objects linking the topology with the geometry and providing an objectification for CSG (Constructive Solid Geometry) operations.

A study on the Structures DD has been recently conducted in [1]. Its true potential has been demonstrated after representing difficult geometries, from reflector antennas to more complicated figures such as satellites. An example

Figure 2.3: Structures Dictionary overview

of a BRep representation of a reflector antenna is shown in figure 2.4. This and others objects have been successfully decomposed in a recent work using the Structure Data Dictionary and, as such, memorized with EDX.

In order to define the dictionaries, a specific language had to be created. This language is called DDL and it will be later discussed in chapter 3. The dictionaries are described in text files and have to be compiled for being used as an input for the EDI library, therefore a lexer and a parser have to be defined for this purpose. The previous version of the DDL was more of a draft and will not be reported in order to avoid confusion.

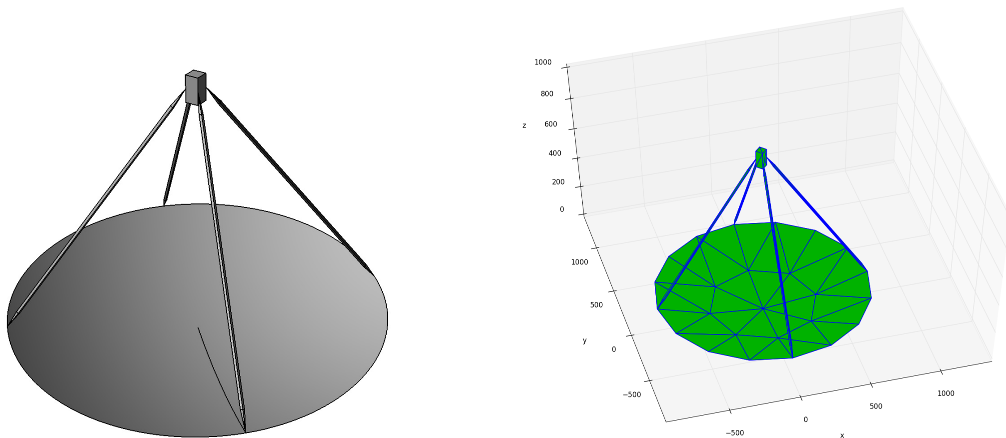


Figure 2.4: Reflector antenna with CAD and with an

2.2 EML

An extension of XML

EML (Electromagnetic mark-up language) is an extension of XML. Basically, in XML the data is stored in a plain text file and the characters in the text file can be distinguished into markup and content. A markup construct beginning with "<" symbol and ending with the ">" symbol is called a tag. Tags come in three flavors: start-tags, end-tags and empty-element tags. An XML element begins at one start-tag and ends at one end-tag of the same type. Among these two tags there is the content of the element. The content can also contain other markup including other elements, which are called child elements. An element can have attributes. Attributes are specified in a start-tag or an empty-element tag. Attributes have a name and a value. More details on the basics of XML can be found in [2].

For a quick understanding of the XML, in figure 2.5 a part of an EML file is reported as an example.

As a specialization EML defines specific tags. In particolar, 4 sections of the document are individuated:

- the header
- the declaration
- the raw data
- the application data section.

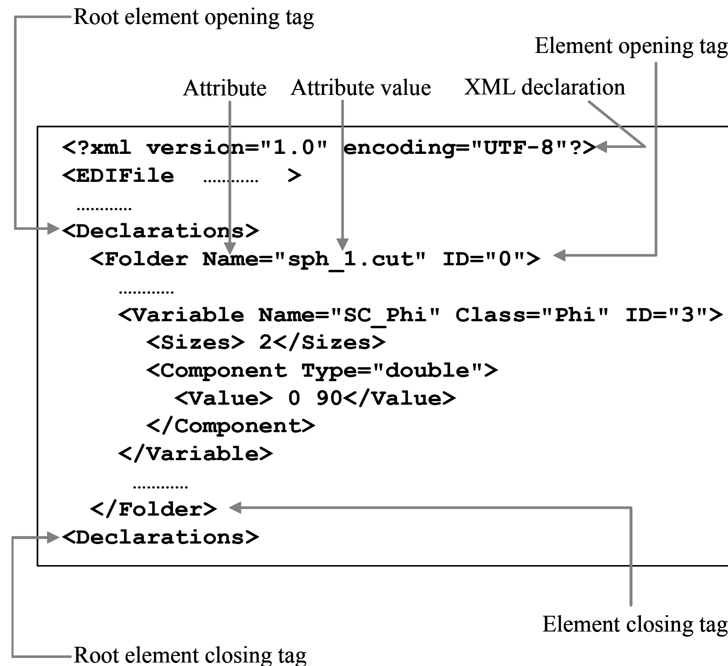


Figure 2.5: EML file in short.

The *Header* specifies the version of EDI, the user and association of the user, the timestamp and the tool used to call the EDI methods.

The *Declaration* section is the place where all the variables go. Variables are grouped in one single folder and are distinguished by ID. They can present *Attribute* tags, with a name and a value, *Domain* tags with the name of the variable used as a domain and *Component* tags. In a similar way components are distinguished by ID, they have a type and they contain values. Variables and components contain a certain number of sizes. Sizes are used to determine the dimension of the domain/codomain. The domains of a variables have in fact their sizes listed in the *Sizes* tag, along with the variable's native sizes. In order to mantain an easily readable declaration section only a few part of the data, if any, is stored there. If the number of values satisfy certain criterions than every value for the entire variable is stored in the raw *Data* section. The *Application Data* is an useful space which contains additional information for the tools that use the file, under a list of *Application* tags, with a name and a content.

For multi-dimensional matrixes of values to be rapresented in a line a convention has to be chosen: the rows of the matrix are listed one after the other within the *Value* tag.

An example of an EMLv1 file is reported below. Take note that, for reasons of space, it is not reported any value at all as long as the type of the components is *void*.

```
<?xml version="1.0" encoding="UTF-8"?>
<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">
<!-- ======================== Header section ======================== -->
<Header>
  <Stamps>
    <Version>EDI Version 1.00.00</Version>
    <Format>XML</Format>
    <DateTime>2011-06-14T00:28:55Z</DateTime>
  </Stamps>
  <Origin>
    <Tool><Name></Name><Version></Version></Tool>
    <Project></Project>
    <User>
      <Name></Name>
      <Affiliation></Affiliation>
    </User>
  </Origin>
  <UserText></UserText>
</Header>
<!-- ===================== Declarations section ===================== -->
<Declarations>
  <Folder Name="EDI_testfile_n2.xml" ID="0">
    <Variable Name="Horn" Class="FarField" ID="1">
      <Sizes> 25 23 21</Sizes>
      <Domain Reference="phi_scan"/>
      <Domain Reference="domain_2"/>
      <Domain Reference="frequency"/>
      <Domain Reference="domain_4"/>
      <Component Type="void">
      </Component>
    </Variable>
    <Variable Name="domain_2" ID="3">
      <Sizes> 23</Sizes>
      <Component Type="void">
      </Component>
    </Variable>
    <Variable Name="freq" ID="2">
      <Sizes> 21</Sizes>
      <Component Type="void">
      </Component>
    </Variable>
    <Variable Name="phi_scan" ID="4">
      <Sizes> 25</Sizes>
      <Component Type="void">
      </Component>
    </Variable>
    <Variable Name="theta_scan" ID="5">
      <Sizes> 27</Sizes>
      <Component Type="void">
      </Component>
    </Variable>
  </Folder>
</Declarations>
<!-- ======================== Data section ======================== -->
```

```
<Data>
  <Variable Name="Horn" RefID="1">
    <Component Type="void">
      </Component>
  </Variable>
</Data>
<!-- ==================== Application Data section ==================== -->
<ApplicationData>
</ApplicationData>
</EDIFile>
```

Listing 2.1: EMLv1 example

More on EML can be found in previous works [3, 4].

2.3 The EDI library

The EDI library is the tool that permits the actual data exchange. It is a software with the role of an IO tool and it can be used as an alternative for proprietary systems. The main software typically is a simulator, a viewer or a data acquisition system and it makes use of the library through an interface.

Using EDI, the software can istantiate variables from the classes defined in the dictionaries and do the actual writing/reading of the EML files. The available back-ends for EDI are a C and a Fortran ones.

EDI has changed radically during the internship period, that's why we will refer to the new version as EDIv2 and to the previous version as EDIv1.

For a better understanding on how it works we can think of EDI as divided in different layers.

- The toolkit level is responsable for the I/O operations, it calls the I/O library whenever it is necessary and it constructs and reads well-formatted EML files. EDIv1 make use of an external library for the XML parsing, Expat.

- Level 0 is responsible for the managing of the variables with their domains and codomains. It contains the functions for changing the header settings and save/load unique files. It permits to modify every property of a variable such as its attributes or the class the variable belong.

- Level 1 adds utilities such as data slicing and sorting. It also furnishes iterators and adds features for incremental I/O.

- Level 2 furnishes already-made aggregates that define currents, fields, geometry elements and so on. From this level, the low-level I/O system is completely transparent.

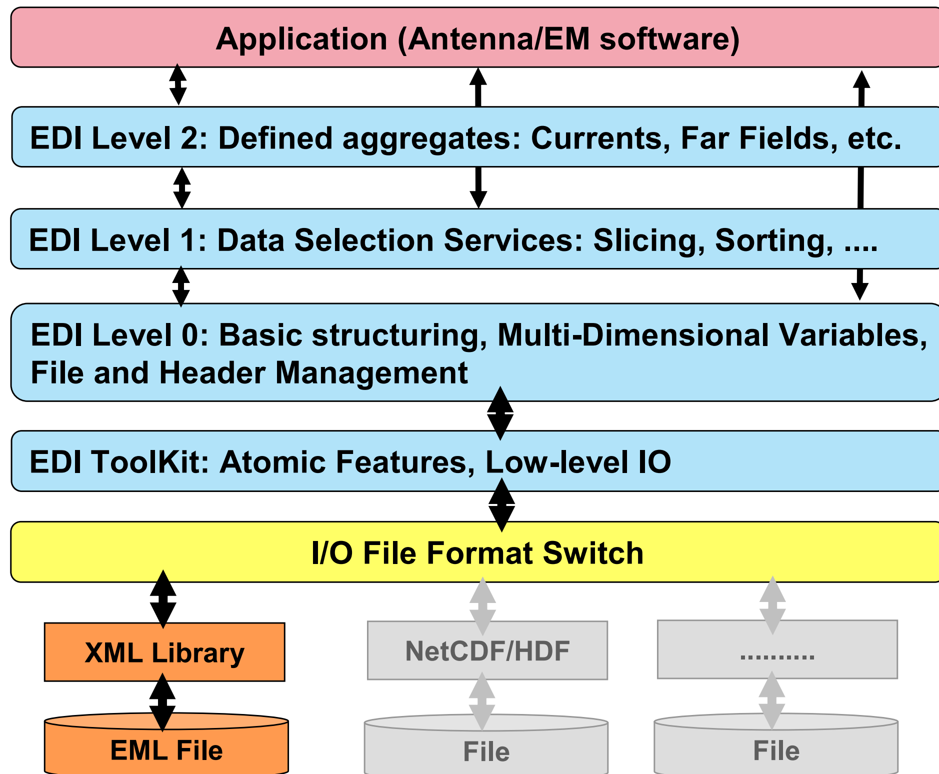


Figure 2.6: The overall structure of the Electromagnetic Data Interface (EDI)

Improvements on the library will focus on the toolkit level and level 0. From now on when talking about EDI, we will refer to just these two levels.

We will now report a short list of methods from the the Fortran interface to show how it looks like. The original manual for the EDI Fortran interface can be found in [5, 6].

EDI_FILE_OPEN(file_id, file_name, status)	Open a specified file and load every section into memory.
EDI_FILE_CLOSE(file_id)	Close the istance of the library and save everything in EML.
EDI_HEAD_QUERY_ORIG_TOOL(file_id, name, name_len, vers, vers_len)	Retrieve the name and version of the tool who used EDI to write the EML file.
EDI_VAR_PUT(file_id, name, rank, sizes)	Insert a new variable.
EDI_VAR_PUT_DOMAINS(file_id, name, rank, sizes, dom_n, doms, dom_len)	Insert the domains of a given variable.
EDI_VAR_REMOVE_ATTRS(file_id, name, count, attr_name, attr_len)	Remove some attributes from a given variable.
EDI_VAR_PUT_COMPONENTS(file_id, name, count, comp_names, comp_len, types)	Insert some components in a given variable.
EDI_VAR_PUT_COMPONENT_SIZES(file_id, name, comp, add_rank, add_sizes)	Set the sizes of a given component.
EDI_VAR_GET_INTS(file_id, name, comp, rank, vstart, vcount, vals)	Retrieve the values of an *integer* component.
EDI_VAR_SET_STRINGS(file_id, name, comp, rank, vstart, vcount, vals, vlen)	Set the values of a *string* component.
EDI_APPDATA_PUT(file_id, tool, data)	Add an entry to the application data list.

Each one of these methods returns an integer value which states if the operation has been completed successfully or not. Each Fortran method is strictly binded to a C method but, considering the differences among the two languages such as the lack of pointers in Fortran, some of them tend to be quite verbose on the parameter list.

EDIv1, including or not its upper levels, is actually used by TICRA for its famous application GRASP [7], which makes use of the FORTRAN interface. ADFEMS, from IDS, another antenna design tool dedicate to space applications, makes instead use of the native C bindings. A MATLAB tool, called EDIFun, has been also built on the latter interface to allow access to EDX files from within this powerful environment. The Fortran interface is also used by a certain number of universities across Europe.

Chapter 3

Language

The best way to define data dictionaries for EDX was by creating an ad-hoc language. This language is called DDL (Dictionary Definition Language) and has been created about ten years ago, as an abstraction to the EML itself.

With the adoption of this language, creating data dictionaries has never been so simple, as the examples will show later.

After the language has been more or less defined, what is left to do is to create the compiler. The quickest way to create one was making use of PLY (Python Lex-Yacc, see [8] for the documentation) witch is an implementation of the lex and yacc parsing tools for Python. Lex and Yacc are a lexical analyzer and a compiler-compiler. They both are open-source freeware [9].

By using these tools we first check the lexicon of the dictionaries and later we check the sintax. The accepted lexicon must be described with the use of a Python dictionary for the reserved tokens and with regular expressions for any general word, number, symbol or a mix of them.

The sintax must be defined by a BNF (Backus-Naur Form) grammar. Every rule of the grammar must be specified within the triple quote, which is usually adopted for the Python documentation, in the first line of a dedicated function for each rule. When the grammar is not ambigous and the parser ends then these functions are executed in the respective order. That means, every time a specific situation has been identified in the input (a new class declaration, a new domain...) the function containing the respective rule will be executed. That function can access to the elements of the lexicon found for the rule and act as a consequence. In our case the parser will produce an output file with extension **.dy** that will be read by EDI.

3.1 Improvements in the DDL

The DDL has drastically changed from the previous versions. The main features introduced consist in the fact that:

- It permits the inclusion of other dictionaries. Now the variables stored in EML files can have references to elements in other dictionaries or folders. This is very important: consider for example of a Surface (topology element) that wants to relate to a geometry element such as a Face, or a Material from the Materials dictionary. Another application is for a reference system that has the origin connected to a physical point.

- It permits the creation of folders and subfolders. Folders are an useful way to organize the classes. It is a self-explanatory concept. Every dictionary will be related to a folder. The EML file will then contain a concealed root folder that will contain every dictionary and every dictionary's subfolder. It is important to notice that, at the moment, it is not permitted for a folder to contain both other folders and classes.

- It introduced subclasses and abstract classes. The inheritance concept is pretty straightforward nowdays and it perfectly matches the definition of classes describing entities. An abstract class cannot be used to create variables but can serve as a base for other classes to be built. Consider a general ScanRange. It can be a 2D or a 3D scan range. Both of them can have different realizations (cartesian, spherical, cylindrical, uv mapping...).

- It introduced prototype versions for folders and classes to preserve their names for a future use.

- It permits to define, along with classes, *members* which are already-made istances of classes ready to be later stored in EDI. Members are unique classes realizations and it is useful to specify some of them along with the dictionaries. For instance, to give predefined names and characteristics for a few typical materials within a Materials class, like PerfectConductor, FreeSpace and so on.

- There are now two types of domains: *domain reference* and *domain index*. The first one uses as dimension of the domain the entire co-domain of another variable. The second one uses the total dimension of the domains and basic sizes of another variable. Variables belonging to a different folder can still be used as a domain.

- The sizes of a variable or a component can be fully or partially specified with a costraint in the number of dimensions or can be completely free of costraints.

- Components can now be nested. Parent components will then assume the type *structure* and can have zero or more children. For example, if a variable has in its codomain 3 electric field components, it can use a single, structured, component which contains the x, y and z axis values.

- For every type of value it is now possible to list a set of choices they can assume and in this way, forbid arbitrary values. Moreover it is possible to define more complex rules to constrain the values also in relation to other components or variables. For the moment, only first order disequations are permitted.

- There are now two special types of components. They are called *component references* and *component indices*: the first ones, given a class, contain pointers to other variables which must belong to that class. The second one, given the name of another variable, contain positions on the domains for that variable. This is basically used to provide a navigation system among variables. Using entire other variables or a position in them as a co-domain grants the user the possibility to relate the variables to each other.

- For every class, a set of rules has be introduced concerning instantiation, presence of elements, and integrity of the data. The instantiation rules basically define how many object of the class can exist at the same time; the presence rules define if a domain/component may be o may be not present in general or according to the presence of other domains/components. In addiction, they specify if there must be at least or at most one domain/component given a set of them. The integrity rules define the ranges for numerical values e.g. strictly positive values.

An example of how the python code is written can be found in the following lines.

```
def p_ClassDeclaration(t):
  '''ClassDeclaration :
      CLASS name PROTOTYPE ENDClassPrototype END
    | CLASS name ADDClass NewClassBodyDeclaration ClassRulesDeclaration ClassMembersDeclaration END
    | CLASS name GETClassName SubClassBodyDeclaration ClassRulesDeclaration ClassMembersDeclaration END
    | CLASS name GETClassName AbsClassBodyDeclaration ClassRulesDeclaration END
    | CLASS name ALIAS QualifiedClassName ADDAliasClass END '''
    global current_class_name
    current_class_name = ''
    global am_i_in_variable
    am_i_in_variable = False
```

Listing 3.1: Use of yacc for the ClassDeclaration rule.

The code is divided in two parts: the part within the documentation quotes defines the grammar rules to be followed, expressed in a BNF notation; the rest of the function is the code that must be executed in case the rule has been recognized during the parsing. Capital letter words are keywords that were defined in the lexer part. For example, the keyword *class* is translated with the token *CLASS*.

In figure 3.1 is reported the equivalent railroad diagram for the rule.

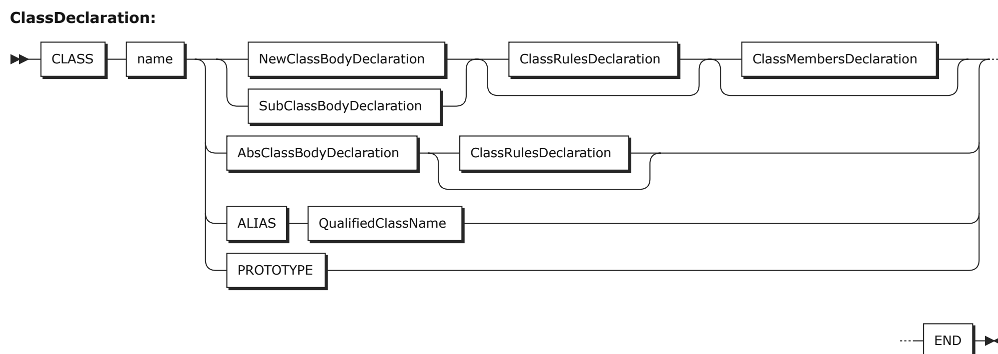


Figure 3.1: Class Declaration railroad diagram.

The actual python code can access the tokens i.e. strings of the rule by using the parameter t which is a Python list. That's not the case of the example because it just delays the analysis to other functions.

The code part of the functions step by step gathers together every aspect of the dictionaries and produce a certain output. The output must summarize the dictionary in a more compact and friendly way to read for the EDI library.

At the moment the behaviour of the output is far from a final form. For the sake of simplicity for now it is just a plain-text file with extension **.dy**. Every line of the dictionary is described by its first characters that will form a *tag*.

```
_DATADICT_|mydict
_FOLDER_|mydict|folder1
_CLASS_|folder1|classname
_CLASS_ATTRIBUTE_|folder1|classname|weather=sun,rain,snow
_CLASS_SIZES_|folder1|classname|2,-,4
_SUB_CLASS_|folder1|classname:subclass1
_CLASS_SIZES_|folder1|classname:subclass1|2,3,4
```

Listing 3.2: Example lines of the **.dy** output file

In the part of the file listed above we can see how the lines are composed. They are separated by the pipe character and the various parts indicate the folder, the class or list of subclasses and relevant informations for the rule. For example, the fourth line indicates that the attribute *weather* of class *classname* can assume only the values *sun*, *rain* and *snow*. The sixth line states that the sizes of the subclass *subclass* must strictly be 2, 3 and 4 as dimensions. This introduces a new constraint from the superclass where the second dimension can assume an arbitrary value.

```
_CLASS_COMPONENT_UNITS_|folder2:innerfolder|simpleclass|simplecomp|aa5^(3-2)/ggg
_CLASS_VALUES_|folder2:innerfolder|simpleclass|simplecomp=1.5,-788.2,0,9,5
_INSTANTIATION_|folder2:innerfolder|simpleclass|singleistance
_PRESENCE_COMPONENT_|folder2:innerfolder|simpleclass|MANDATORY|simplecomp
_PRESENCE_COMPONENT_|folder2:innerfolder|simpleclass|ATLEASTONE|simplecomp,simplecomp2
_PRESENCE_COMPONENT_|folder2:innerfolder|simpleclass|MANDATORY|simplecomp|IF|attrname=blue
_PRESENCE_DOMAIN_|folder2:innerfolder|simpleclass|ATLEASTONE|dom,otherdom
_PRESENCE_DOMAIN_|folder2:innerfolder|simpleclass|MANDATORY|dom|IF|attrname=blue
_PRESENCE_COMPONENT_|folder2:innerfolder|simpleclass|MANDATORY|simplecomp|WITH|DOMAIN=dom
_PRESENCE_DOMAIN_|folder2:innerfolder|simpleclass|MANDATORY|dom|WITH|COMPONENT=simplecomp
_BOUNDS_|folder2:innerfolder|simpleclass|simplecomp|[4,6.5];[-1,1];
```

Listing 3.3: Example lines of the **.dy** output file

In the example of code 3.3, the first line specifies the units for a certain component and the second line present the limited amount of values it can assume. The third line introduce the fact that the class can only be instantiated once. In the rest of the lines we can see how the policies for the presence of an element can vary from mandatory to conditional ones.

In the last line we can see how bounds are described. In this case the values of the component *simplecomp* can only be contained in intervals $[4, 6.5] \cup [-1, 1]$.

3.2 Improvements to the existing Dictionaries

In this section parts of the existing Fields data dictionary, omitting the rules, are reported. This is the most simple dictionary to show as the Structures data dictionary is far more complicated and it is actually being revisioned, nonetheless it has been treated more in details in other places [1].

This dictionary is meant to define three main classes:

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

A theorethical approach and decomposition for this dictionary can be found in [10]. At the end of this section, In figure 3.2, is reported the main design of the Fields DD.

In this first piece of the dictionary we can see how every main component that a near field can assume is listed. Some of them may be optional and some mandatory.

For the moment, the dictionary did not make use of the import instruction to include other dictionaries but we can assume that in the future the coordinates will be linked to physical points.

```
data_dictionary Fields

folder Fields

class Field
  abstract
  sizes -
end

class Near extends Field
  attribute TimeTypeAxis : Time Frequency
  attribute SpaceTypeAxis : Space Wavenumber
  attribute TimeDependency : +j\omegat
  sizes 1
  component Frequency sizes 1 type reference class Frequency end
  component Time sizes 1 type reference class Time end
  component ScanRange sizes 1 type reference class ScanRange end
  component BeamPointing sizes 1 type reference class BeamPointing end
  component Projection sizes 1 type reference class ProjectionComponents:3D end
  component EE sizes 1 type reference class FieldComponents:E end
  component HH sizes 1 type reference class FieldComponents:H end
  component dEdXi sizes 1 type reference class FieldDerivatives:dEdXi end
  component dEdt sizes 1 type reference class FieldDerivatives:dEdt end
  component dEdOmega sizes 1 type reference class FieldDerivatives:dEdOmega end
  component dHdXi sizes 1 type reference class FieldDerivatives:dHdXi end
  component dHdt sizes 1 type reference class FieldDerivatives:dHdt end
  component dHdOmega sizes 1 type reference class FieldDerivatives:dHdOmega end
  component PowerNormalisation sizes 1 type reference class PowerReference end
  component PhaseReference sizes 1 type reference class PhaseReferencePoint end
  component RelativeGainOffset sizes 1 type reference
                                class RelativeGainNormalizationOffset end
  component EllipticalPolarizationParameters sizes 1 type reference
                                class EllipticalPolarizationParameters end

end #class
...
```

Listing 3.4: Part of the Fields data dictionary.

In this second piece of dictionary we can see how the field derivatives can be specified all together in a *FieldDerivatives* class listing every necessary domain. Each one of them has a different measurement unit, that is the reason why the components are specified in subclasses. That is true for both electric and magnetic fields components.

```
class FieldDerivatives
  abstract
  domain Frequency reference Frequency
  domain Time reference Time
  domain ScanRange reference ScanRange
  domain BeamPointing reference BeamPointing
  domain ProjectComponents reference ProjectComponents
  sizes 1
end #class
class dE_dXi extends FieldDerivatives
  component dE_dXi sizes 1 type dcomplex units V/m^2 end
end #class
class dE_dt extends FieldDerivatives
  component dE_dt sizes 1 type dcomplex units V/(ms) end
end #class
class dE_dOmega extends FieldDerivatives
  component dE_dOmega sizes 1 type dcomplex units Vs/m end
end #class
class dH_dXi extends FieldDerivatives
  component dH_dXi sizes 1 type dcomplex units A/m^2 end
end #class
class dH_dt extends FieldDerivatives
  component dH_dt sizes 1 type dcomplex units A/(ms) end
end #class
class dH_dOmega extends FieldDerivatives
  component dH_dOmega sizes 1 type dcomplex units As/m end
end #class
```

Listing 3.5: Part of the Fields data dictionary.

In this third part of the dictionary we can see the inheritance applied to the *ScanRange* classes. Scan ranges can be both bidimensional or tridimensional. Here we see severals coordinate systems applied to the 3D representation. It is now easy to specify in a *component reference* only a certain 3D scan range class in such a way that the variable or variables it refers to is/are coherent with the rest of the data.

```
class ScanRange
  abstract
  sizes -
end #class

class _3D abstract extends ScanRange end

class Cartesian extends ScanRange:_3D
  component x sizes 1 type double units m end
  component y sizes 1 type double units m end
  component z sizes 1 type double units m end
```

```
end #class

class Spherical extends ScanRange:_3D
  component theta sizes 1 type double units deg end
  component phi sizes 1 type double units deg end
  component r sizes 1 type double units m end
end #class

class Cylindrical extends ScanRange:_3D
  component phi sizes 1 type double units deg end
  component z sizes 1 type double units m end
  component ro sizes 1 type double units m end
end #class

class AzimuthElevation extends ScanRange:_3D
  component Az sizes 1 type double units deg end
  component El sizes 1 type double units deg end
  component r sizes 1 type double units m end
end #class

class ElevationAzimuth extends ScanRange:_3D
  component alpha sizes 1 type double units deg end
  component epsilon sizes 1 type double units deg end
  component r sizes 1 type double units m end
end #class

class uv extends ScanRange:_3D
  component u sizes 1 type double end
  component v sizes 1 type double end
  component v sizes 1 type double units m end
end #class

class CartesianWavenumber extends ScanRange:_3D
  component kx sizes 1 type dcomplex units rad*m^-1 end
  component ky sizes 1 type dcomplex units rad*m^-1 end
  component kz sizes 1 type dcomplex units rad*m^-1 end
end #class
```

Listing 3.6: Part of the Fields data dictionary.

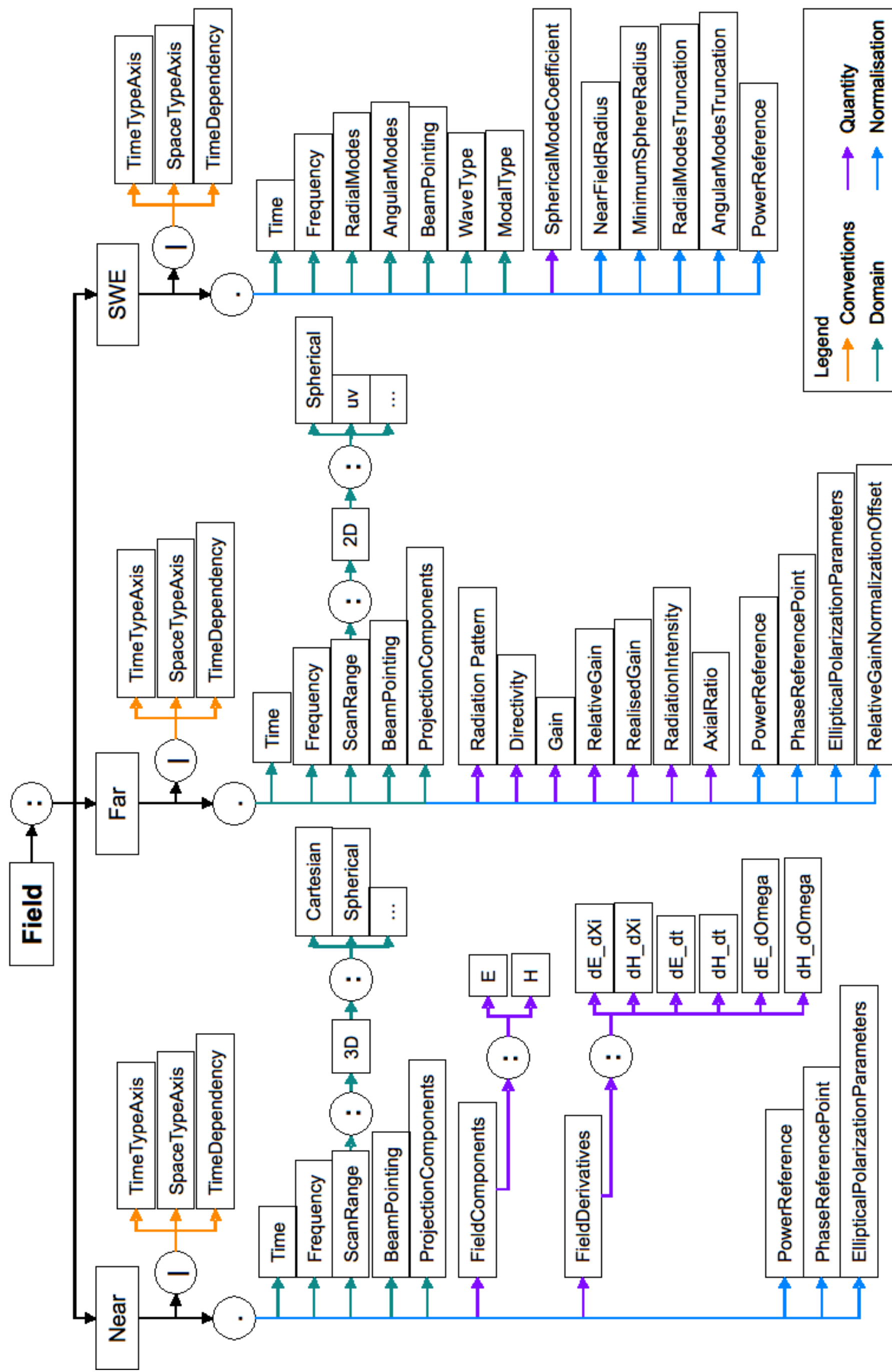


Figure 3.2: Fields Data Dictionary basic design.

Chapter 4

Library

The library for the actual exchange of EDX files is called **EDI** which is an acronymous that stands for Electromagnetic Data Interface. As said in the previous chapters, there is an existing EDI library which is now being replaced with a new version, main focus of the internship activity. Again, to avoid confusion we will refer to the old version as EDIv1 and let the new version be EDIv2. EDIv2 was born for mainly two reasons: to update the type of structures that can be stored and fix a certain amount of bugs present in the old version. In addiction to this, it has been added a new part with the role of checking if the meaning of stored variables respect the rules defined in their dictionaries. Retrocompatibility with the C and FORTRAN interfaces has been maintened altough a few differences are present. In version 2 a Python interface has been introduced that make use of the C interface using *ctypes*, a Python library that provides C compatible data types, and allows calling functions in DLLs or shared libraries [11].

4.1 Improvements in EML

The changes in the DLL described in the previous chapters must reflect to the EML. The previous version wasn't sufficient to describe every new aspect. That is the reason why even EML has been updated to a new version.

In the *Header* section, at the same level as *Version*, *Format* and *Date-Time* tags, a new tag called *Capacity* has been introduced. This tag has two attributes that define two thresholds. The first one is the limit to the amount of values that a variable can mantain in the declaration section. After such a limit is reached, every value of the variable is then stored in the *Data* section. That is because the shorter and easily readable the *Declaration* section section is, the better. The second one is the limit to the amount of values that

a variable can mantain in the *Data* section. After that threshold, the values are supposed to be store in binary files. This has not been implemented yet in EDIv2 and will be discussed in the future developments chapter.

The *Usertext* tag can no more contain different lines but just an indefinite long string. This feature has been removed because it was never used so far and seemed therefore redundant, as all mainstream programming languages allow embedding of end-of-line characters within a string.

The declaration section can now host multiple folders. Folders can even be nested and have an optional attribute *Dictionary* which relates them to the dictionaries. Folders have an unique global ID that goes in [0-99] and folders at the same level are granted to have consecutive IDs. The IDs of the variables are dependent from the folder they belong. Every folder can contain at most ten thousand variables and their unique and global IDs go from [0-99][0001-9999] where the leftmost digits are their folder's ID. For example, the variables in a folder with ID 3 can assume IDs from 30001 to 39999. The *Class* attribute of each variable is now mandatory. Anyway, for a first period this checking, along with the checking of the class rules, can be disabled.

In the previous version the sizes of the domains were listed together with the ones of the variable. Now there is no more the need to explicitly store the sizes of dependent variables, as EDI quickly computes them avoiding potential size conflicts. What is left are just addictional dimensions for the variable.

As stated in the previous chapter two types of domain now exist. Their tag is the same, what changes is that they have an unique *Name* attribute that fully identify them and they present a *Reference* attribute or an *Index* attribute. These two assume the full path of another variable. A full path of a variable is the string containing the path to the folder/subfolder they belong, separated with a double colon symbol, and the name of the variable itself. An example of these two types of domain can be found in the following code.

```
<Domain Name="fr" Reference="Field::Frequency"/>
<Domain Name="phi" Index="Field::Phi"/>
```

Listing 4.1: Example of the new domains in EMLv2.

The type of the components is now stated as an attribute of the *Component* tag along with the name and the ID. Components do not have global IDs but are univocally identified by a variable ID and their own ID (unique in the variable). Components at the same level within a variable have consecutive IDs.

New types for the components have been introduced, according to the changes in the DDL. The *Type* attribute when describing a *component reference* is the same as a simple *string* but, if it is the case of a reference, a new attribute called *Range* will appear. The same happens for the *component index* case where the type is an integer. The two cases are infact a collection of strings and a collection of indices.

This solves several issues with the management of references in the old version and most of all permits to save more than one of them. Of course, the strings representing every class or variable in the *Range* attribute must be full paths. They can be given to the API as pure names of local variables or classes (i.e. in the same folder) but they will be converted to full paths when stored.

Sizes for the variables and the components must now always appear. Even if they are not adding dimensions. In that case they'll be a vector of just one element with value 1.

The *Value* tag for complex numbers can now present an optional attribute, *Format*, with options *Plane* and *Nice*. *Plane* is the classic method for storing complex numbers: it is just the listing of the real and the imaginary part one after the other. *Nice* is a more estetic and more simple to read for certain parsers way of saving the values using parenthesis and commas. For EDIv2 files the second version is the default value of the format attribute.

```
<Value>(0,2) (3,4) (1,0)</Value>
<Value format="plane">0 2 3 4 1 0</Value>
```

Listing 4.2: Examples of the format attribute.

About IDs, every time EDIv2 reads an EML file it will reassign them (possibly mantaining the same values when last closing the file).

The last changes regard the *Data* section. Variables are now divided in folders as well. The only folders they belong in this section, anyway, are the children of the root folder i.e. the dictionary folders. We will explain later the reason of this behaviour.

Folders in the *Data* section contain as attribute only their ID. Inside the folders every variable of any subfolder is listed. In the old version of EML their attributes were the variable's name and the variable's ID in a RefID attribute. Everything that remains now is just their ID in an attribute simply called *ID*. This also applies to their components and nested components.

4.2 Improvements in the library

The EDI library is the most complicated part of EDX. The main features of the new version are:

- Reading and writing EML files. Since version 2, EML files are well-defined but, in order to maintain retrocompatibility, EDIv2 is also able to read and write old versions.
- Keeping in memory the overall structure. Everything that has been read from the file or has to be written must be kept in memory all at once.
- Furnish a complete and documented interface to interact with the data. EDIv2 presents an updated C interface, a mantained Fortran interface and a brand new Python interface. During the internship period a Python interface has been created for EDIv1 as well.
- Perform the checking on the meaning of the data. A new module of the library, given a compiled dictionary, validates every variable for that dictionary before writing the EML file.

The EDI library has been written in C/C++ version 11 and is perfectly compilable under GCC and Visual Studio 12. It has been tested under the most common operating systems such as the most recent versions of Windows, GNU/Linux and Mac OS X under both 32 and 64 bits architectures.

The reason why using C++ instead of other languages is because of the performances, the modernity and the popularity of the language and of its standard libraries. There are two reasons why using C in some parts. The first is because the library that handles the XML uses a C approach: C pointers, characters basic types and so on. The second is because of the interface that EDI is going to furnish. Infact, C can interact far better than C++ with other languages, for example with *ctypes* used in Python.

The exception system has not really been defined. For the moment EDI uses assertions in a restrict set of critical cases. Generally, when an error occurs the return parameter of the C functions is used as a flag. The list of errors is documented and a function called *edi_error()* also exists to retrieve the error string description from the flag value.

The library is structured in classes as we can see from figure 4.1.

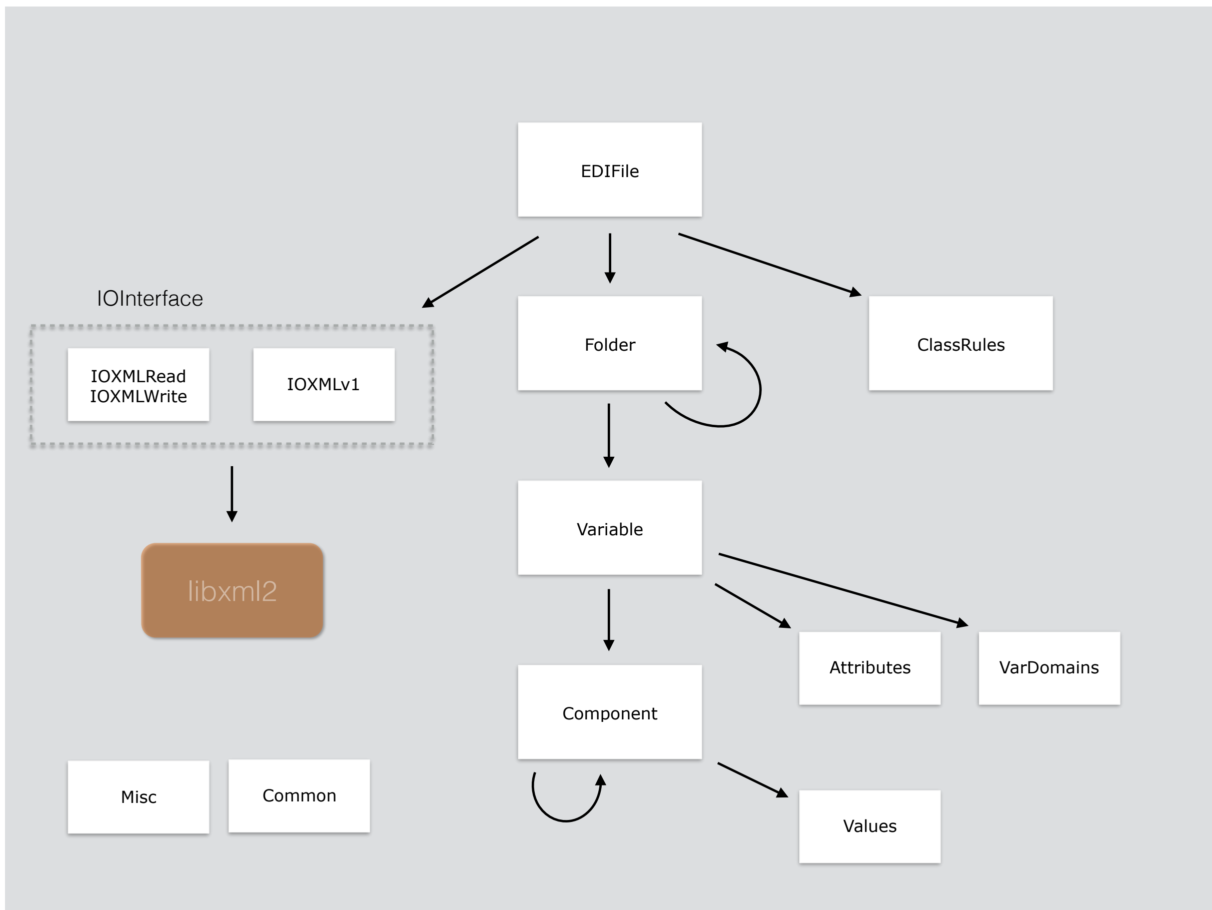


Figure 4.1: Basic structure of EDIv2

The most important class is *EDIFile*. A major difference with the previous version is that the library can now be instantiated more than once. In the past just one process had to be open and this process did manage more than one file at the same time using different ID namespaces. Now, different processes are created and are identified with an ID usually called in the documentation as *file_id*. EDIFile contains a map with the IDs of every Folder, Variable or Component to provide a quick access. As the hierarchy suggests, it also contain one folder, called Root. The root folder behaviour is completly transparent to the absolute paths of the variables and it is used just for management simplicity. Every *Folder* object will contain a vector of other folders and a vector of other variables. As for the variables it has been decided to separate their contents among other C++ classes such as *Attributes* that is self-explanatory and *VarDomains*. This last one contains several methods to find other variables used as domains and correctly update

the sizes of the variable they belong. This is true for both domain references and domain indexes.

It is very important to notice that if a variable used as a domain does not exist yet the dependent variable is disabled. In the case a variable is disabled it is not possible to use its values or to save the EML file. The only way to enable again a variable is to solve every one of the domains or remove them. It is not necessary for the user to indicate to EDI that a new variable inserted should resolve a pendent domain because it automatically and periodically looks for the domains resolution.

Every *Variable* contain a vector of *Components* objects with all their properties. A *Values* class has been created to make transparent some of their aspects such as the checks for the indicization with the dimensions of the component. Every *Component* inherits the sizes from its parent and in the case of a top component it will inherit from the variable sizes. The updating of every sizes vector is automatic and it can delete some data. This issue will be faced in future releases. For example, if we want to restrict the lenght of a physical object a domain size will change. Consequently the components of the dependent variable will change their inherited sizes as well but we may want to delete just a slice of the data and not it all.

Another branch that comes from the main class is the one for the IO. The IO can now be performed in both EML v1 and v2. The version of existing files is recognized automatically and it can be changed at will. Obviously an error will occurr in case new features of EDIv2 are going to be saved in the old EML version.

The new IO can now save the output to different files. The most important difference among the IO performed in EDiv1 and the new version one is the library they make use for XML. The old version uses the Expat XML parser for reading and as for the writing it writes directly on the file. This behaviour is really dangerous for a security and integrity point of view. For version 2 a completely different library has been decided to be used. Libxml2 is a software library used in a lot of opensource projects. It is written in C and is highly portable since it depends on standard ANSI C libraries only and it is released under the MIT license. With Libxml, EDIv2 can perform the reading and the writing for both the EML versions. Libxml provides two types of parsing: SAX and DOM. SAX is a bit faster for the reading because it is event-based. At the end, for the sake of simplicity it has been decided to use DOM. More details on Expat and Libxml can be found in [12] and [13].

The *Misc* class contains several useful methods for the library to use such as the conversion from strings of characters to vectors of numbers. The *Common* class contains the definition of the error codes, the enumerators for the types, for the tags and policies and so on.

Finally, the checking is performed inside the *ClassRules* class. It happens whenever the user wants to save the output file. The idea of a rule is generally identified with the Rule interface as we can see in figure 4.2. The interface has its own general properties such as the folder associated with the dictionary, the tag identifying the rule and a boolean method that states if the rule is satisfied or not.

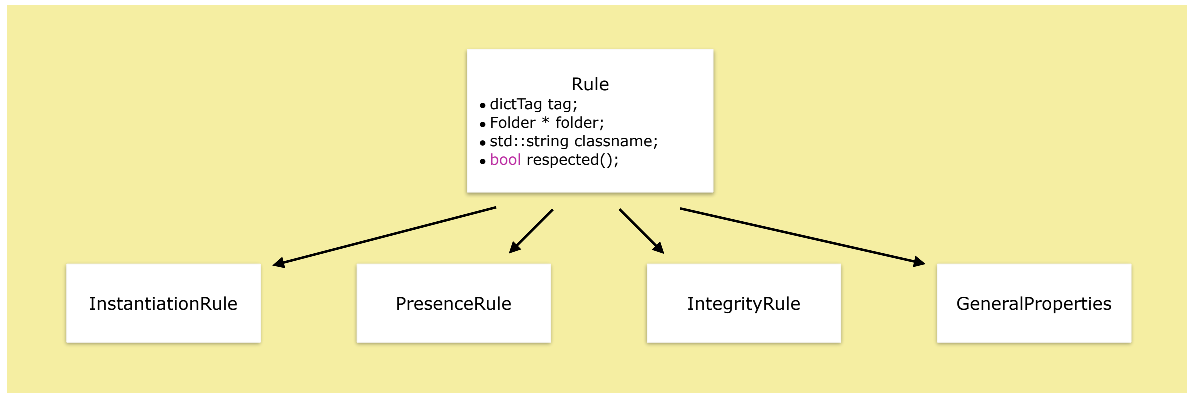


Figure 4.2: Rules inheritance in EDIv2

Every concrete rule class extends the *Rule* interface and, when instantiated, knows exactly the folder and class it must check. When the checking is performed a simple function retrieves the variables belonging to the class and a series of conditions concerning the presence of elements, the content of the elements, the allowed intervals for the values and so on start to be valuated. At the end, if one of the rules returns a negative feedback the user is notified of the error and the saving of the file is not performed.

The reading of the dictionary files associated with the folders goes in the *EDIFile* class but the analysis of their lines is done in the *ClassRules* object. If an include rule for the dictionary is found the library will look for a homonymous **.dy** file in the same filesystem's folder to add.

The dictionaries to look for are a property of the folders but, for the moment, single rules or entire dictionaries can also be added manually from the interface.

In a first trial period the entire checking system of EDI can be disabled from the interface using the method *edi_ignore_rules()*.

It is important to notice that the C++ realization of the methods for the C and Fortran bindings are slightly different and for this reason are placed in different folders of the project. That is because of a simple but tricky aspect: Fortran does not support pointers. Every vector used as parameter

by Fortran must be pre-allocated and there will be additive parameters that serve as flags to signal when the buffers are too small. Using the C binding, on the other side, implies to free the pointers from outside EDI. Fortunately, the Python layer does it automatically but in any case EDI furnish a pair of methods to free different kind of vectors.

Along with the new features, a set of new methods for the interface have been added. The most significative ones are reported below.

```c
/* Save the changes to a specified EML file and close the session.*/
int edi_file_close_other_file(edi_id file_id, char const * const other_file);
/* Load a dictionary given its filename.*/
int edi_load_dictionary(edi_id file_id, char const * const dict);
/* Add a single rule.*/
int edi_class_add_rule(edi_id file_id, char const * const rule);
/* Ignore the class rules for the current session.*/
int edi_ignore_rules(edi_id file_id);
/* Get the capacity thresholds for the declaration and data section. */
int edi_head_get_capacity(edi_id file_id, int * const dec, int * const data);
/* Set the capacity thresholds for the declaration and data section.*/
int edi_head_set_capacity(edi_id file_id, int dec, int data);
/* Get a list of IDs and names for the subfolders of a given folder.*/
int edi_folder_list(edi_id file_id, edi_id folder_id, size_t * const
    list_size, edi_id ** const ids, char *** const folder_names);
/* Remove a subfolder given its name and the parent folder.*/
int edi_folder_remove(edi_id file_id, edi_id folder_id,
    char const * const folder_name);
/* Get the ID of a subfolder given its name and the parent_folder.*/
int edi_folder_query(edi_id file_id, edi_id folder_id, char const *
    const folder_name, edi_id * const sub_folder_id);
/* Add a subfolder given its name and retrieve the ID.*/
int edi_folder_put(edi_id file_id, edi_id folder_id, char const *
    const folder_name, edi_id * const newid);
/* Set the data dictionary of a given folder.*/
int edi_folder_set_data_dictionary(edi_id file_id, edi_id folder_id,
    char const * const data_dictionary);
/* Add a variable given its name and class. Retrieve the variable's ID.*/
int edi_var_insert(edi_id file_id, edi_id folder_id, char const *
    const var_name, char const * const class_name, edi_id * const
    newid , int rank, int const * const sizes);
/* Add a new domain reference.*/
int edi_var_add_domain_ref(edi_id file_id, edi_id var_id,
    char const * const dom_name, char const * const dom_ref);
/* Add a new domain index.*/
int edi_var_add_domain_index(edi_id file_id, edi_id var_id, char const *
    const dom_name, char const * const dom_index);
/* Return a list of the domain references for a given variable.*/
int edi_var_list_domain_refs(edi_id file_id, edi_id var_id,
    char ** const names, char ** const doms);
/* Add a new component reference given the parent variable or component.*/
int edi_var_add_component_reference(edi_id file_id, edi_id var_id,
    edi_id parent_comp_id, char const * const name,
    char const * const classname, int * const id);
/* Add a new component index given the parent variable or component.*/
int edi_var_add_component_index(edi_id file_id, edi_id var_id,
    edi_id parent_comp_id, char const * const name,
    char const * const index, int * const id);
```

Listing 4.3: Some of the new methods added to the C interface.

The Python interface is really close to an 1 on 1 to the C interface. As said before, it has been realized with *ctypes*. There are a lot of advantages of using Python but the main one is its simpleness. As an interpreted language it does not need to be compiled and can provide a really fast testing for the entire library. Nonetheless its data structures are well known for being handy. We will now show a simple example of how to use the Python binding for creating a variable dependent from another one.

```
edi = EDIwrapper() #open an istance of EDI
err, status = edi.edi_file_open("file_in.edx") # choose an empty file
edi.edi_load_dictionary("Fields.dy")
# functions are as easy to use as they look
edi.edi_head_set_capacity(25, 2000)
edi.edi_head_set_orig_tool('python testing tool', '-')
edi.edi_head_set_usertext('my text')
edi.edi_ignore_rules() # just for testing purposes
#create two folders
err, f1 = edi.edi_folder_put(0, "first folder")
err, f2 = edi.edi_folder_put(0, "second folder")
#create a frequency variable in the first folder
err, v1 = edi.edi_var_insert(f1, "frequency", "Frequency", [10])
err, c1 = edi.edi_var_add_component(v1, -1, "freq", "double", "MHz")
edi.edi_var_set_doubles(v1, c1, [0, 0], [10, 1],
[2.400, 2.405, 2.410, 2.415, 2.420, 2.425, 2.430, 2.435, 2.440, 2.445])
#create a power variable in the second folder
err, v2 = edi.edi_var_insert(f2, "power", "PowerReference", [1])
# the frequency variable is a domain of the power variable.
edi.edi_var_add_domain_reference(v2, "frequency", "first folder::frequency")
err, c2 = edi.edi_var_add_component(v2, -1, "Radiated", "double", "W")
# the dimension of the components are dictated by the frequency domain.
edi.edi_var_set_doubles(v2, c2, [0, 0, 0], [10, 1, 1],
[5.482, 5.483, 5.497, 5.495, 5.502, 5.493, 5.493, 5.495, 5.486, 5.483])
# save the output in another file and close the instance of EDI
edi.edi_file_close_other_file("file_out.edx")
```

Listing 4.4: Python usage of EDI.

As the library was increasing and becoming more populated of methods and layers the tests started to become more difficult as well. At some point fixing a problem could easily cause creating a new one. That is why an automatic test for the most basic actions has been created. The automatic test makes use of Python and consists in a set of checks that will have a positive or negative result. If any problem is found it is consequently being reported along with some details. The output of the automated test is a simple html page with a table in it describing the attempts and the results.

An example of the table is reported in figure 4.3 where the failure was introduced on purpose.

Of the entire EDIv2 code a documentation has been generated using Doxigen. Doxigen is a free GNU software that builds website-like or LaTeXdocumentation

TEST	RESULT	DETAILS
File opening	Correct	
Different sessions	Correct	
Duplicate folders insertion	Correct	
Not-existing folders removal	Correct	
Folder insertion in deleted folder	Correct	
Variable insertion in deleted folder	Correct	
Duplicate variables insertion	Correct	
Not-existing variables removal	Correct	
Attributes editing	Correct	
Attributes removal	Correct	
Insertion of domains that still don't exist	Wrong	The domain has not been solved
Propagation of a change in sizes for a domain	Correct	
Propagation of a change in the component sizes for a domain reference	Correct	
Propagation of a variable removal for a domain	Correct	
Duplicate component insertion	Correct	
Not-existing component removal	Correct	
Component insertion in not-existing variables	Correct	
Component insertion in not-existing components	Correct	
Removal of a parent component remove the nested components too	Correct	
Editing of an application data entry	Correct	
Not-existing application data removal	Correct	
File closing	Correct	

Figure 4.3: Output table for the automated test.

for projects written in several languages.

4.3 Retrocompatibility

In order to mantain an high level of retrocompatibility two main issues had to be faced.

The first one is the EML. The previous version was quite mature but never well-defined. As new features had been added to the DDL, the structure of the EML file was not able to describe them. Nonetheless, it was too soon for the old EML version to be declared obsolete. That is why it has been decided to mantain the possibility not only to read old versions of the files but to write them as well. This is now transparent to the user of EDI, apart from the fact that the checking for classes and variables cannot be performed.

Infact, the classes for the variables were not mandatory and there was only one folder. A certain number of tests were conducted to see if every old file could be read and converted to the new version. The conversion worked successfully for every example case used as a meter.

The second aspect was about the interface. In respect to that, not only a set of new methods were added, but some had to change. We'll not be listing every little change in every method but we can at least identify the common changes. The thing that changed the most is the use of the IDs. As stated before, EDIv2 now creates one instance of the library for every file being open. It is then fundamental to gain access to the correct instance of EDI with the *file_id*. The methods of the interface that didn't have it before were the ones regarding the declaration and data section. That is because the IDs of the variables were unique among every file. This is not true anymore so the integer value of the *file_id*, furnished during the creation of the instance, has to always be passed as the first parameter. A second main change that regarded only the C interface, but not the Fortran one, is the use of C structures as parameters. For example, when returning the list of the variables, the return type was a pointer to a C structure containing an array of strings, an array of dimensions and the number of strings. Everything in one single structure. Even if *ctypes* could handle easily C structures, it has been decided that such a complication wasn't necessary at all.

Chapter 5

Conclusions and future developments

As a conclusion, we can state that rewriting the library from scratch was worth the effort. Many new features have been added quickly as the library was being redesigned. Especially the Python part has introduced a really powerful and dynamic testing system.

The language part is now under revision and has already been subjected to a few changes due to the recent work of Step Over Srl on the Structures Data Dictionary. It is now more clear and engaging.

In any case, several other improvements are yet to be done. First of all, the transaction for existing software to adopt the new version of EDI. It is a matter of changing the function calls but some issues may still come up. Another tool that is going to pass to the new version is the EDIFun tool for MATLAB but, fortunately, it is based on the Fortran interface so the problems to be faced are quite the same.

A part from the usual assessments for when a new software is created (speed, stability, consistency), a few other points to focus will be:

- The slicing system. As discussed before, components that are redimensioned lose their values even if their domains decrease in length. It can be useful to avoid this and mantain the values instead of reinitialize them.

- The dimensions system. For an external user it may be difficult to understand how many sizes are concatenated. That is why sizes with just one dimension with value 1 should have the possibility to be omitted during the usage.

- The **.dy** dictionaries format. There are far better formats for the dic-

tionaries instead of lines on a text file. After having decided the format to use, the validation methods in EDI have to change as a consequence. The output of the Python parser must change accordingly.

In addiction, by the fact that the total number of dictionaries is not high, they can be included within the library itself avoiding the necessity for the user to compile them with Python.

- The last and most important change is the introduction of binary files for the storing. Basically XML files are text files and, because of that, a conversion has to always be performed for numerical values. The worst aspect is that they occupy a lot of space in the hard drive. It is not reasonable to mantain billions of values in a unique XML file. That is why a new system for storing the data is taking place. In the future three approaches will be available: to use just plain EML, to use a zipped EML file thanks to a contribution of an archive manager and to use again an archive approach but with binary files. The last method consists in mantaining the EML file but all the variables whose values exceed the second threshold stated in the *Capacity* tag will not be saved in the data section but in a different, binary, file making use of HDF5. There will be one HDF5 file for each dictionary. HDF5 is a file format designed to store and organize large amounts of numerical data [14]. It has a hierarchical structure which applies well with the EML system. The EML file and the HDF5 files will be zipped together in a unique **.edx** file. The University of Aachen developed a first experimental HDF5 I/O interface for EDIv1, showing very good compression and access speed [15]. It needs to be consolidated and integrated into the new version of EDI.

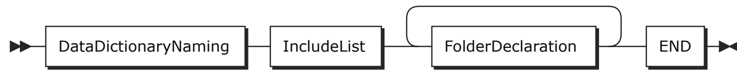
During the period of the internship a lot of things concerning EDX have changed drastically and a lot of others have been put under discussion. The DDL is starting to be more valuable and is now sensed by the users thanks to the introduction of the validation system. The Data Dictionaries are constantly being refined by the means of frequent confrontations among partners and new ideas coming up. The source code of EDI has been modernized, documented and is now more approachable from an external point of view.

Appendix A

BNF and Railroad diagrams for the DDL

```
DataDictionaryDeclaration ::= DataDictionaryNaming IncludeList (FolderDeclaration )+ END
DataDictionaryNaming ::= DATA_DICTIONARY name
IncludeList ::= (INCLUDES name (',' name) * |)
FolderDeclaration ::= FOLDER name (PROTOTYPE | (ClassDeclaration )+ | (FolderDeclaration )+ ) END
ClassDeclaration ::= CLASS name (( NewClassBodyDeclaration | SubClassBodyDeclaration )
  > (ClassRulesDeclaration|)(ClassMembersDeclaration|) | AbsClassBodyDeclaration
  > (ClassRulesDeclaration|)| ALIAS QualifiedClassName | PROTOTYPE ) END
NewClassBodyDeclaration ::= (OVERRIDE name | NEW | ) PropertiesDeclaration (ComponentsDeclaration | )
SubClassBodyDeclaration ::= EXTENDS QualifiedClassName (PropertiesDeclaration |) (ComponentsDeclaration |)
AbsClassBodyDeclaration ::= ABSTRACT (EXTENDS QualifiedClassName |) (PropertiesDeclaration |) (ComponentsDeclaration |)
PropertiesDeclaration ::= AttributeDeclaration * DomainDeclaration * (StructureDeclaration |) SizeDeclaration
AttributeDeclaration ::= ATTRIBUTE name ':' name (',' name) *
DomainDeclaration ::= DOMAIN (name |) (REFERENCE | INDEX) QualifiedClassName
StructureDeclaration ::= STRUCTURE (CARTESIANPRODUCT | LISTOFTUPLES)
SizeDeclaration ::= SIZES (NONE | (number | '-')+ )
ComponentsDeclaration ::= (UnnamedComponentDeclaration | MultipleComponentDeclaration)
UnnamedComponentDeclaration ::= COMPONENT ComponentContentDeclaration END
MultipleComponentDeclaration ::= (COMPONENT name (ComponentContentDeclaration | PROTOTYPE) END)+
ComponentContentDeclaration ::= (SizeDeclaration | ) (TypeDeclaration (UnitsDeclaration |)
  > (ValueOptionsDeclaration |) | TYPE ( STRING (ValueOptionsDeclaration |)| STRUCTURE
  > ComponentsDeclaration | (REFERENCE | INDEX)CLASS QualifiedClassName (ValueOptionsDeclaration |)))
UnitsDeclaration ::= UNITS ISOUnitsDefinition
ISOUnitsDefinition ::= (name | number |symbol)*
TypeDeclaration ::= TYPE (VOID | BOOL | CHAR | INT | FLOAT | DOUBLE | COMPLEX_TYPE | DCOMPLEX)
ValueOptionsDeclaration ::= VALUES ((number)+ | (name)+)
QualifiedClassName ::= name ('::' name)* '::' name (':' name)*
ClassRulesDeclaration ::= RULES (InstantiationRule|) (PresenceRules|) (IntegrityRules|) END
InstantiationRule ::= INSTANTIATION SINGLEISTANCE
PresenceRules ::= PRESENCE PresenceRuleDecl + END
PresenceRuleDecl ::= (DOMAIN | COMPONENT) ((MANDATORY | OPTIONAL) (name | ConditionDeclaration) |
  > FORBIDDEN ConditionsDeclaration | (ATLEASTONE | ONLYONE) name (',' name)* )
ConditionsDeclaration ::= name (WITH (DOMAIN | COMPONENT | IF name IS) name) END
IntegrityRules ::= INTEGRITY IntegrityRuleDecl + END
IntegrityRuleDecl ::= COMPONENT name (BOUNDS Bound (';' Bound)* | RELATION RelationalExpression) END
Bound ::= ('(' | '[') number ',' number (')' | ']')
RelationalExpression ::= ('<' | '>')('=' |) number
ClassMembersDeclaration ::= MEMBERS (VARIABLE name VariableBodyDeclaration END)+ END
VariableBodyDeclaration ::= AttributeDeclaration * DomainDeclaration *
  > StructureDeclaration SizeDeclaration (ComponentsDeclaration |)
```

DataDictionaryDeclaration:



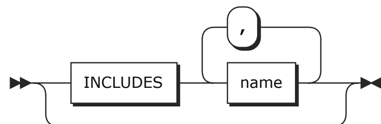
no references

DataDictionaryNaming:

referenced by:

- DataDictionaryDeclaration

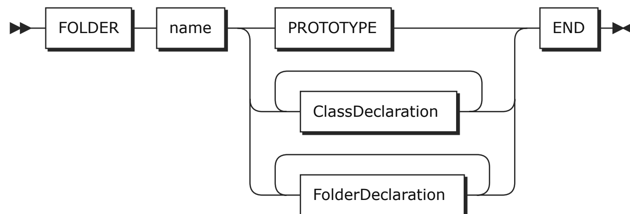
IncludeList:



referenced by:

- DataDictionaryDeclaration

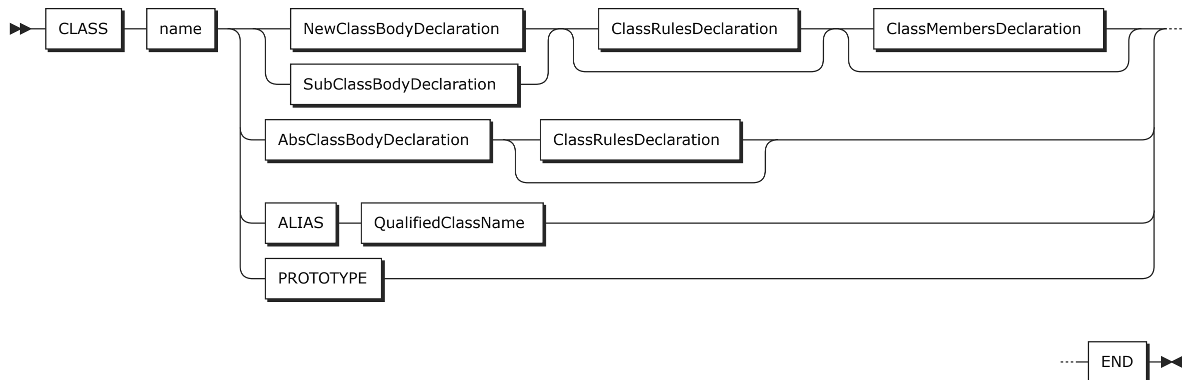
FolderDeclaration:



referenced by:

- DataDictionaryDeclaration
- FolderDeclaration

ClassDeclaration:



referenced by:

- FolderDeclaration

NewClassBodyDeclaration:

referenced by:

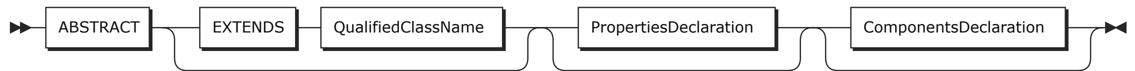
- ClassDeclaration

SubClassBodyDeclaration:

referenced by:

- ClassDeclaration

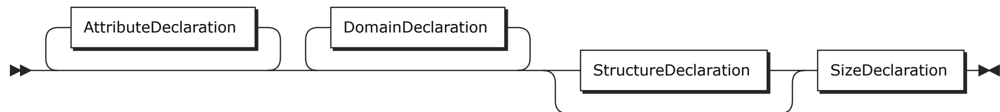
AbsClassBodyDeclaration:



referenced by:

- ClassDeclaration

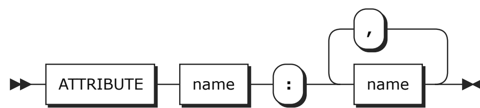
PropertiesDeclaration:



referenced by:

- AbsClassBodyDeclaration
- NewClassBodyDeclaration
- SubClassBodyDeclaration

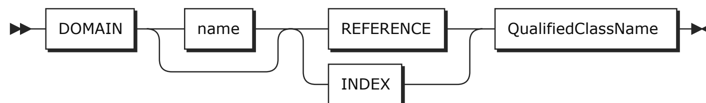
AttributeDeclaration:



referenced by:

- PropertiesDeclaration
- VariableBodyDeclaration

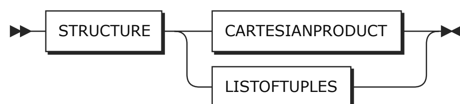
DomainDeclaration:



referenced by:

- PropertiesDeclaration
- VariableBodyDeclaration

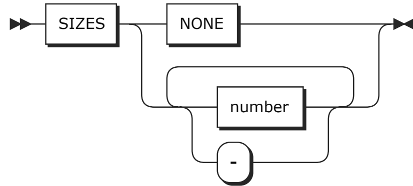
StructureDeclaration:



referenced by:

- PropertiesDeclaration
- VariableBodyDeclaration

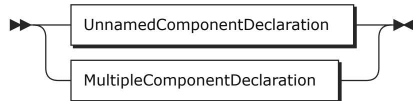
SizeDeclaration:



referenced by:

- ComponentContentDeclaration
- PropertiesDeclaration
- VariableBodyDeclaration

ComponentsDeclaration:



referenced by:

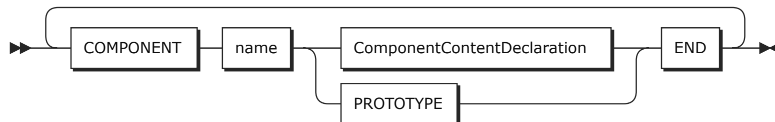
- AbsClassBodyDeclaration
- ComponentContentDeclaration
- NewClassBodyDeclaration
- SubClassBodyDeclaration
- VariableBodyDeclaration

UnnamedComponentDeclaration:

referenced by:

- ComponentsDeclaration

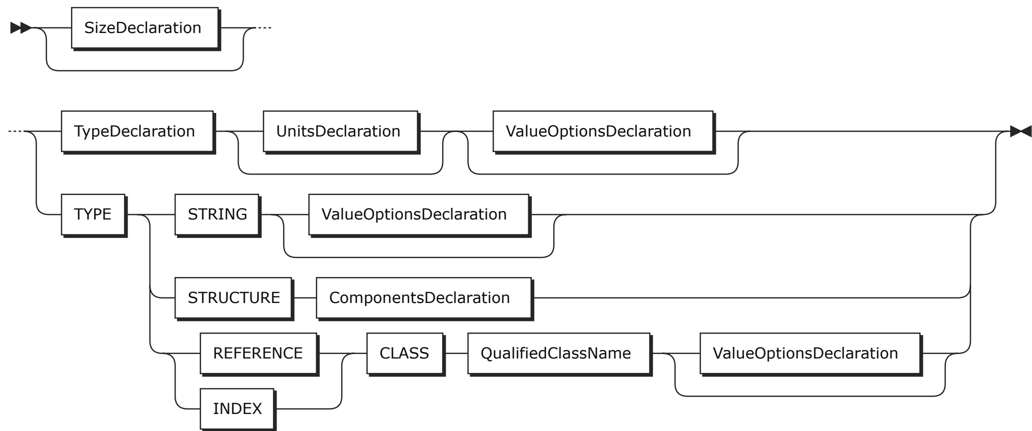
MultipleComponentDeclaration:



referenced by:

- ComponentsDeclaration

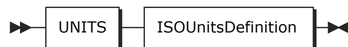
ComponentContentDeclaration:



referenced by:

- MultipleComponentDeclaration
- UnnamedComponentDeclaration

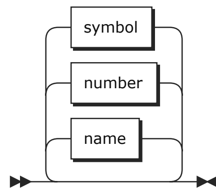
UnitsDeclaration:



referenced by:

- ComponentContentDeclaration

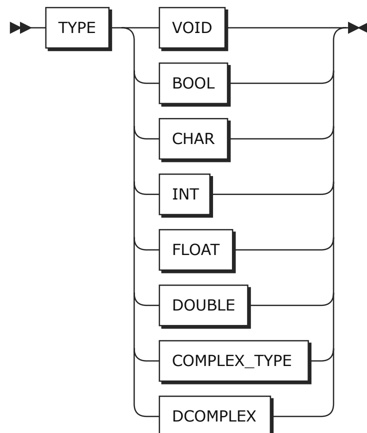
ISOUnitsDefinition:



referenced by:

- UnitsDeclaration

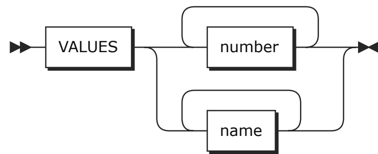
TypeDeclaration:



referenced by:

- ComponentContentDeclaration

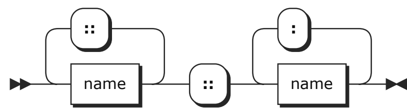
ValueOptionsDeclaration:



referenced by:

- ComponentContentDeclaration

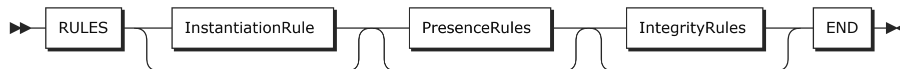
QualifiedClassName:



referenced by:

- AbsClassBodyDeclaration
- ClassDeclaration
- ComponentContentDeclaration
- DomainDeclaration
- SubClassBodyDeclaration

ClassRulesDeclaration:



referenced by:

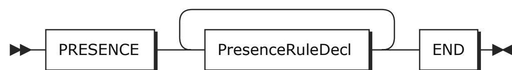
- ClassDeclaration

InstantiationRule:

referenced by:

- ClassRulesDeclaration

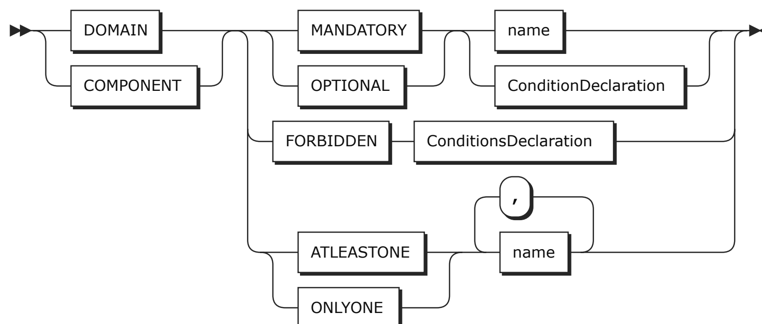
PresenceRules:



referenced by:

- ClassRulesDeclaration

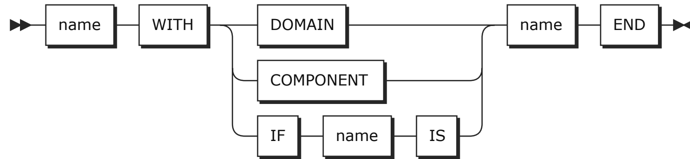
PresenceRuleDecl:



referenced by:

- PresenceRules

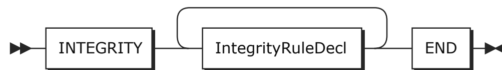
ConditionsDeclaration:



referenced by:

- PresenceRuleDecl

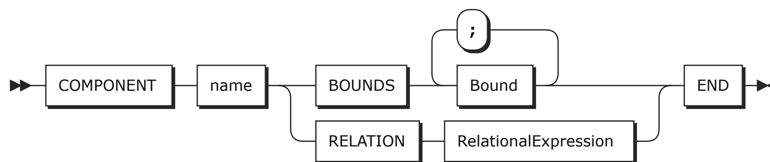
IntegrityRules:



referenced by:

- ClassRulesDeclaration

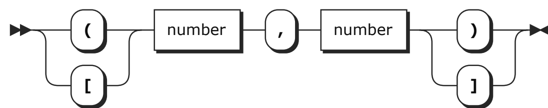
IntegrityRuleDecl:



referenced by:

- IntegrityRules

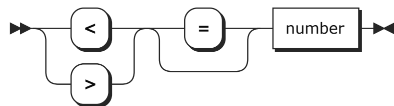
Bound:



referenced by:

- IntegrityRuleDecl

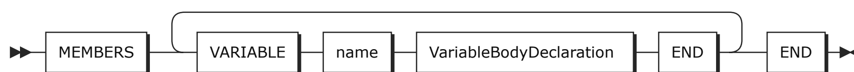
RelationalExpression:



referenced by:

- IntegrityRuleDecl

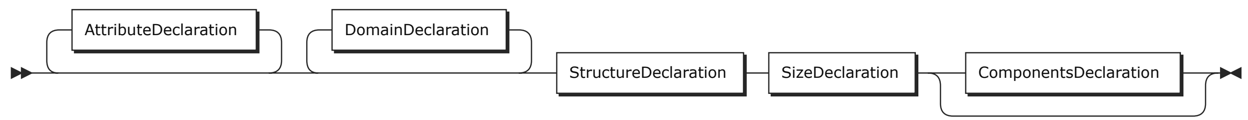
ClassMembersDeclaration:



referenced by:

- ClassDeclaration

VariableBodyDeclaration:



referenced by:

- ClassMembersDeclaration

... generated by Railroad Diagram Generator

Bibliography

[1] F. Rossi, "Data model and software prototype for antenna geometrical information." University of Padua, 2015.

[2] http://www.w3.org/XML/.

[3] P. E. Frandsen and M. Sabbadini, "An introduction to the Electromagnetic Data Exchange language."

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

[5] F. Silvestri and M. Ghilardi, "Electromagnetic Data Interface, FORTRAN User Manual - EDI Level 0 and Level 1," 2007.

[6] F. Silvestri and M. Ghilardi, "Electromagnetic Data Interface, FORTRAN User Manual - EDI Level 2," 2007.

[7] http://www.ticra.com/products/software/grasp.

[8] http://www.dabeaz.com/ply/.

[9] J. R. Levine, T. Mason, and D. Brown, *lex & yacc (2 ed.)*. O'Reilly, 1992.

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

[11] https://docs.python.org/2/library/ctypes.html.

[12] http://www.xmlsoft.org/.

[13] http://expat.sourceforge.net/.

[14] https://www.hdfgroup.org/HDF5/.

[15] M. Dirix, "Intermediate Report EAML VII," 2014.