EDX

An introduction to the Electromagnetic Data Exchange language

February 2009

Issue 3

Poul Erik Frandsen[1] and Marco Sabbadini[2]

[1]*TICRA, Laederstraede 34, 1201 Copenhagen K, Denmark, Email: ticra@ticra.com*
[2]*European Space Agency, Keplerlaan 1, 2201AZ Noordwiijk, The Nederlands, Email: marco.sabbadini@esa.int*

TICRA Ref.: S1326-14
ESA Ref.: EWP-2345

An introduction to the Electromagnetic Data Exchange language

Written by Poul Erik Frandsen, Marco Sabbadini
Graphics by Poul Erik Frandsen, Marco Sabbadini
Printed by ESTEC Reproduction Services

First edition April 2008
Second issue October 2008

TABLE OF CONTENTS

1. Introduction

The Electromagnetic Data Exchange (EDX) language system has been developed in the EAML and ACE projects [1, 2]. Although not accepted by any formal body, it is the goal for it to become a reference for data exchange among electromagnetic modelling software tools, at least in Europe. The EDX background and the requirements are covered in a number of reports and conference papers to which the interested reader is referred [5,7,8,11,18].

The diversified nature of electromagnetic modelling techniques and the quest for the best modelling solutions for each problem encountered, make it necessary to use different tools in combination to take advantage of the most efficient method on each part of an antenna system at different stage of its design and development cycle. The exchange of data among the software tools is therefore a very important aspect of antenna modelling, both for research activities and for industrial use, and the primary approach used for this purpose is the exchange of data in files.

Most antenna modelling tools use proprietary data formats, while they all handle the same physical entities: field, currents, geometry, and transforming data from one format to the other or writing functions to do so is a costly chore. It is quite evident that a common data format, widely accepted within the antenna community, would lead to much easier interchange of data, with the potential of making a much wider range of modelling options available to all its members.

The need for common way to describe physical objects and quantities involved in the electromagnetic modelling of antennas has been stressed by many people at least for the past 20 years or so. Actions to fulfil this need have been promoted since the early 1990's [5]. More recently the MADS project, funded under the EU-FP5 programme, has produced a first draft for a common format, the Electromagnetic Data Format (EDF). Unfortunately, the limited resources available resulted in limited flexibility and the absence of a plane-text data file option. To overcome these limitations the Antenna Software Initiative of the Antenna Centre of Excellence (an EU-FP6 Network of Excellence) and of the European Antenna Modelling Library team, working under ESA contract, have joined forces to develop the Electromagnetic Data Exchange language.

This report is intended as an introduction for those who will actually integrate the EDX in their software for day to day use. In section 2 the three main components of the EDX are introduced. In the sub-sequent sections the three main elements of the EDX are discussed and it is shown how the system can be integrated in existing software. This can be done in a quite simple manner or more advanced solutions can be made depending on the required level of flexibility.

2. Data exchange basics

The Electromagnetic Data Exchange language is devoted to simplify the transfer of information among electromagnetic modelling tools with the additional requirement to use a form which is also convenient for the human user.

Information transfer has been studied in detail for many decades and there are plenty of model from which is possible to draw inspiration when attempting to define a new language for a certain class of data, for instance the ISO/OSI model for communication protocol [3]. Since the objective of EDX is to allow the transfer of information among different parties, the data to be handled with EDX are static, i.e. there are not supposed to be manipulated in the form used they have. Of course, the closer they are to a form suitable for processing the better, as this minimises the cost of reading and writing them, but compromises going in the opposite direction are to be accepted from the outset when dealing with a language to exchange data among very diverse antenna modelling and measurement systems.

A very basic example is the fact that different programming languages store matrices with different ordering of the indices and only one can be used as reference for the language. Incidentally, EDX is structured in such a way that this particular aspect is invisible to a developer using the I/O library that implements it. Still if the indices have to be reordered there is inevitably a small performance penalty with respect to the case where no reordering occurs.

There main elements are essential to fully define the data exchange protocol for static data: meaning, structure and layout, roughly corresponding to the two top layers of the ISO/OSI model: application and presentation. The remaining five, dealing with the detail of data transport from source to user, are left to the computing platform by using the facilities offered by them.

The *meaning* of data is probably the most difficult part, it is usually very well understood but it is also quite difficult to explicitly formulate it as this entails attention to a lot of details which are often implicitly assumed, while for transparency and consistency it is important to have them completely spelled out in the definition of the language.

The *structure* of data reflects the physical background of data, which dictates the best way to organise them to make sure they can be easily understood by humans and handled, as easily as possible, by computer programs.

The *layout* of data requires serious thinking to select a flexible and robust solution, including the selection of a storage format to put them on computer files or other digital storage and transmission means. However the power offered by information technology makes it possible to implement very flexible and elegant solutions at this level with a relatively small effort.

Figure 2.1 illustrates how these three elements can be accommodated in a relatively simple structure. Meaning is captured by using appropriate naming and structuring the information in different levels, each capturing the specifics of one element. For example, the presence of an element named `ScanRange` in the `Field` object is used to show that the latter is a function of the first, while at the next level the `ScanRange` is declared to be spherical and to have three elements, two being angles and the third a distance.

The use of multiple levels also introduces flexibility. It is quite easy to imagine changing the scan range from spherical to rectangular by just pointing to a different definition. Therefore it is also possible to extend at a later stage the number of possible scan range types by just adding new ones without compromising existing data sets.

Finally the structure also suggests a layout. Translating the boxes into independent blocks of data and labelling them with their names produces a very readable layout. Each element can also be given a name and the link to lover levels can be put next to it.

In the artificial intelligence domain this type of systematic representation of the elements of a specific domain of knowledge is called *ontology*. The building rules well known and they are quite similar to the rules governing human languages and the communication of meaning between human beings (linguistic and semiotic). These rules together with the practical objectives of a data exchange facility have been kept well in mind while designing and implementing EDX.

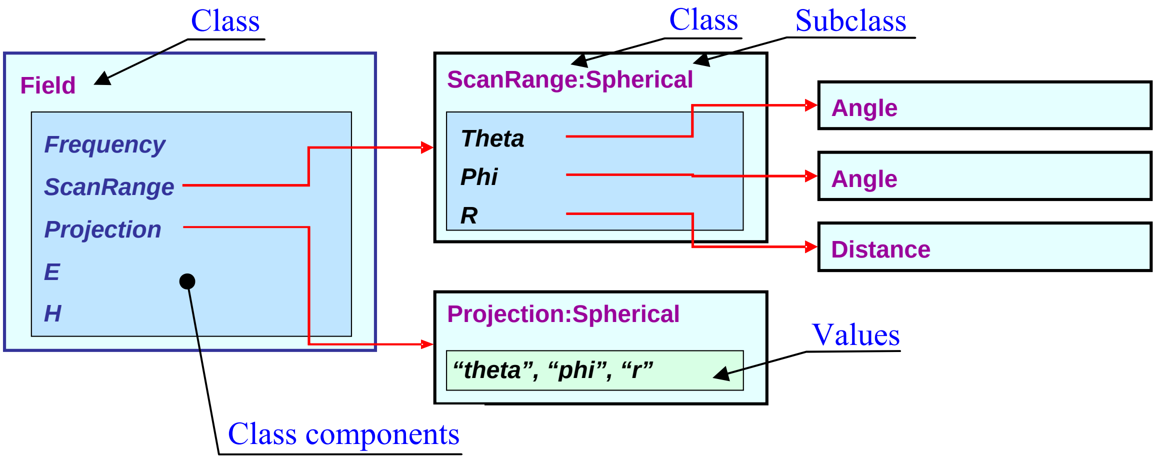


Figure 2.1 – The content of a simple data set to exchange field data

3. An outline of the EDX

When approaching the development of a data exchange language, three key questions must be considered:

- What data shall be handled?
- How shall the data appear in the file?
- How shall software tools access these files?

Since the EDX language should answer these questions on behalf of a rather large community of scientists and engineers, it was felt necessary to lay down a small set of basic requirements that could be shared by many. The EDX development has been the following tenets:

1. Capability of handling of most common data sets in the field of Antenna and Electromagnetic Engineering.
2. Possibility to tailor and extend the data sets without breaking the validity of data files.
3. Easy to understand human-readable data files option.
4. Flexible data access to support the different needs of a wide variety of algorithms, tools and design procedures.
5. Ability to handle multiple representations of the same physical or mathematical quantity.
6. Possibility to store meta-data (author, date, comments) as well as special private data required by some tools.
7. Openness to future revisions and extensions and robustness to withstand the changes involved.
8. Possibility to implement an open and freely available library for access to data.
9. Possibility to handle large amounts of data via interfaces to high performance standards.
10. Platform independence –with respect to both software and hardware.

Many robust data exchange solutions compatible with these requirements are available thanks to the large effort devoted worldwide to this crucial issue of the information technology era. Selecting the right starting point already requires a significant ingenuity. Past experience within the antenna community has shown that a simple format based on a line-by-line description of the file contents would be severely limited. At the same time, very flexible formats having programming-language-like data constructors have proven to be rather difficult to use for the average developer, while high-performance solutions, like NetCDF [6] or HDF5 [10], pose some accessibility problems for non-expert users. The need for human-readable files combined with that of handling large amount of data lead to the choice of XML (eXtensible Mark-up Language) [4], the leading reference for text-based data exchange, for small and medium sized data sets with the option to used high-performance formats for large ones.

An XML-based format satisfies all the above requirements, but number 9, and benefits of a well-established and very general framework, so it is an almost perfect candidate. Unfortunately a direct use of XML requires starting with an agreement on the representations of all quantities involved. Further analysis of the needs showed that it would have been impossible to produce such common representation in an acceptable timeframe. Building on the positive experience made with NetCDF in the EDF development, it was then decided to devise a simple, yet sufficiently flexible and open, language based on XML able to support an incremental growth and consolidation of the definitions. This also offered the opportunity to comply with requirement 9, by ensuring that the information carried by the new language would be compatible with existing high-performance data exchange standard, like NetCDF and HDF5.

Most often the data to be exchanged, like a tabulated antenna pattern, are a series of numbers with no specific meaning on their own. Their meaning needs to be conveyed separately, either using a fixed format specified by some document or by attaching this information to the data. Since this information is the most relevant for the human reader, EDX goes one step beyond. It starts by specifying the meaning and structure of data and then gives the data. A neutral, quite general and simple grammar, easily expressed in XML, has been developed to describe the content of data files, the Electromagnetic Mark-up Language (EML). It features the basic elements required to represent generic electromagnetic quantities, like the ability to specify scalar, vector and matrix quantities and to associate measurement units to them.

EML only provides the tools to specify how the information is conveyed, e.g. how different quantities are called or and structured, but it does not specify names and structures. Clearly a common data exchange language requires that also these elements are agreed. One major strength and weakness of natural language is that misunderstandings occur very easily just because the meaning of words and sentence structures cannot be strictly codified. A set of Electromagnetic Data Dictionaries (EDD’s) establishes the lexicon of the exchange language, i.e. how to convey the actual meaning using the EML grammar. For example, the possible types of component projections and sampling grids for a far field are uniquely defined to make information exchange about antenna patterns possible.

Finally, a software library, the Electromagnetic Data Interface (EDI), allows standardised access to data in the EDX language. Of course, having the EML and the EDD’s in place would be sufficient to guarantee successful data exchange. Still everybody would have to write functions to read and write the EDX files, with an enormous duplication of effort, simple mistakes making the actual exchange difficult and, last but not least, an open door to “slightly twist” the language to fit some specific purpose. A software library relieves developers from the burden to write their own access functions, avoids mistakes and provides a common baseline to which any other implementation can be compared for compliance with the EDX reference.

The structure of EDX can then be summarised using the following symbolic formula:

EDX = EDD’s + EML+ EDI

Furthermore a set of utilities, the EDX Companion Tools, is under development to help the use and extension of EDX.

3.1 The meaning of data – the Data Dictionaries

In a Data Exchange Standard the Data Dictionary defines the meaning of data and the conventions for their exchange. The data dictionary defines *exactly and in detail* all the elements that shall appear in a data set.

Note, that the term “data set” is used – not “files”. A data set is a collection of related simple data elements which as a whole models a physical entity such as a field, a bicycle tire or any other physical item. The data set may be in a file, it may be transmitted over a channel or it may reside in a memory area. For the time being it is assumed that a file only contain one data set. In section 4 it will be clear that the EDX enables several data sets to appear in the same file by using an approach similar to directories or folders in computer file systems.

While EML defines the grammar, a data dictionary contains and defines the ‘words’ to be used to describe the main physical entity, be that a field or a bicycle tire. In other words, it establishes the precise meaning of the data found in the data set, how they are represented numerically and the conventions adopted for their exchange; For instance, the implied time dependence in spectral representations (e.g. $e^{+j\omega t}$).

A physical entity is seldom atomic. Therefore each data set contains a number of elements, each one having its own definition and representation. Often these lower level entities appear in different data sets. The most obvious example is frequency; another one are reference systems. It is worth noting that in most cases these lower level entities have a more abstract nature that he main one,

The definition of a data dictionary is a lengthy and rather complex matter – especially if many do participate to it as required for a common reference language. To streamline the definition of EDX, the whole ‘universe’ of antenna modelling quantities has been separated in homogeneous ‘sub-universes’, like fields, currents, geometries, which are the object of separate data dictionaries. This choice raises the issue of coherence across the different dictionaries, for example ensuring that the frequency quantity has the same definition in all of them. At the same time it is necessary to ensure that the physical and mathematical meaning associated to a given name is univocally defined across the entire ‘universe’ [5]. A global data dictionary is used for this purpose.

Six data sets have been initially identified. 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.

Currently the lexicon required for fields (near, far and spherical wave expansion) and required for currents and meshes has been fully defined [12,13], while a draft exists for the global dictionary involving all the six data sets listed above.

3.2 The form of data – the EML

The definition of the language grammar is a very delicate step as it has a huge impact on how the quantities defined by the Data Dictionaries will be described in practice and may actually pose severe limits to it.

An extensive analysis was made [7,9] leading to the identification of two fundamental and very common data structures: n-dimensional matrices, to represent nD-tensor fields, and hierarchical compounds, to represent the logical and lexical connection among data. The grammar selected to describe their structure is inspired to NetCDF [6] and, as already said, it is a specialisation of XML.

The key idea is that a single data entity, the variable, is sufficient to the static representation of data stored into files. A variable may or not depend on other ones, via its domain, thus allowing the representation of sampled functions of n variables. A variable may have multiple components, each one of them being multi-dimensional. For example, an electromagnetic field sampled on a plane at multiple frequencies can be represented as:

```
Variable emField
 Domain frequency
 Domain samplingX
 Domain samplingY
 Component E (complex, dimension=3, unit=V/m)
 Component H (complex, dimension=3, unit=A/m)
end
```

A variable may also have no domains and its components may just be references to other variables, allowing the creation of hierarchies of aggregates. For example:

```
Variable aDouble ReflectorAntenna
 Component feedArray
 Component subReflector
 Component mainReflector
end
```

Note that, to make them more immediate, the two examples above do not use the actual EML grammar but a simplified version of it. The detailed discussion in section 4 is based on the actual grammar and complete description of it can be found in [4].

The examples above highlight an important feature of EDX, if an extra domain or component is added to a variable, a tool not knowing about it is still able to use the rest of the variable data, just ignoring the extra information. In the same way it is possible to add extra variables to a data set without compromising its usability from unaware tools. This is a fundamental improvement with respect to usual situation.

EML files are organized in four main sections: Header, Declarations, Data and Application Data. The Header section contains general data like author, date, EDI version, etc. The Declarations section contains the definition of all variables as well as the values for small-sized ones. Larger amount of numerical data are found in the Data section, arranged in blocks, one for each of variable. Finally, the Application Data section is available for tools that need to save private data.

3.3 The handling of data – the EDI

The third element of EDX is a software library providing all functions required to access data. The Electromagnetic Data Interface is a relatively small library (a few thousands lines) written in C++ and equipped with application programming interfaces in C++, FORTRAN90 and MATLAB® (only Level 2 at the moment).

The library has a layered structure with each layer offering more advanced features compared to the lower one.

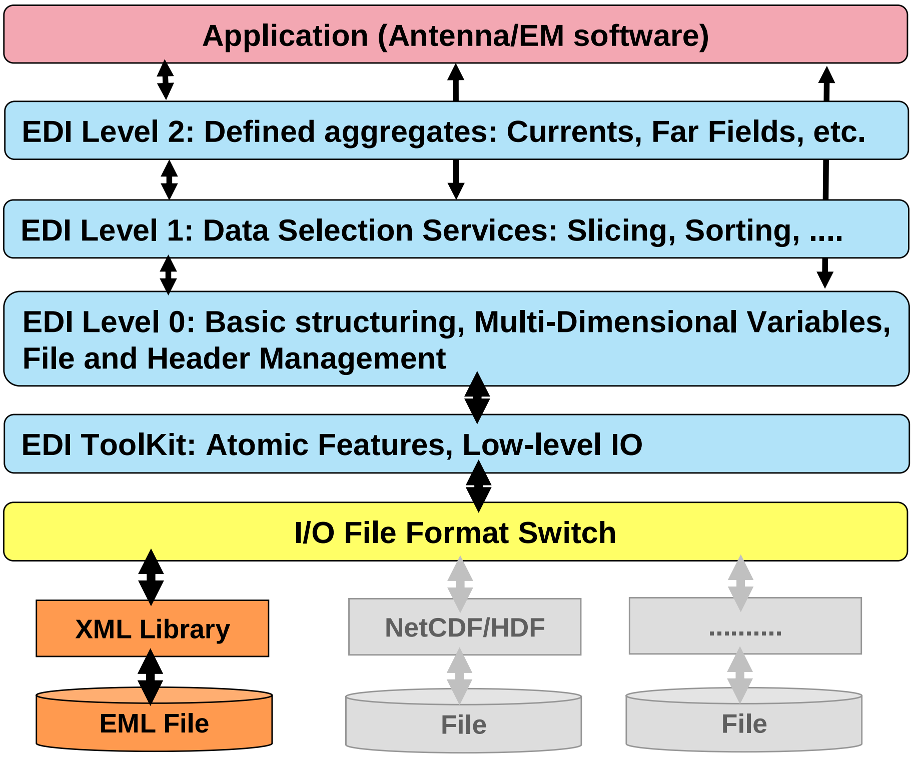

Application (Antenna/EM software)
EDI Level 2: Defined aggregates: Currents, Far Fields, etc.
EDI Level 1: Data Selection Services: Slicing, Sorting, ...
EDI Level 0: Basic structuring, Multi-Dimensional Variables, File and Header Management
EDI ToolKit: Atomic Features, Low-level IO
I/O File Format Switch
XML Library
NetCDF/HDF
....
EML File
File
File

Figure 3.1 – The overall structure of the Electromagnetic Data Interface (EDI)

The main purpose of EDI is to simplify writing computer programmes using EDX, the highest level offers a single call access to complete data sets (although with very limited functionality) and more and more detailed access is available working at lower levels. The EDI Toolkit is a foundation layer and it is not accessible through the application programming interface.

As clearly shown in figure 3.1 all functions offered by the library are independent from the actual format in which the data file is written, so that different ones can be used with no changes in the modelling tool.

3.4 EDX Companion Tools

On top of the tree core elements described so far, it is the intention to offer a number of complementary utilities to enhance the use of EDX. These utilities are being developed and include:

1. EDX Tool: an I/O function generator
2. A generic EDX data browser
3. A generic EDX data visualisation tool
4. A validation tool for EDX implementations.

Furthermore the possibility to develop a CAD data import filter associated to a Geometry and Materials Data Dictionary is being explored.

The EDX tool is a code generator that takes as input a formal definition of a data dictionary, written in a language very close to EML, and a set of programming language fragments corresponding to the different constructs of the language and produces I/O functions for the data dictionary quantities by assembling and completing the fragments.

The Generic EDX data browser allows the navigation through data stored in the EDX language. The data organisation and values are visualised in text form in a way that makes them easy to understand for human users with a good knowledge of antenna engineering and possibly no knowledge of computer science and software engineering.

The Generic EDX data visualisation tool displays and prints the data stored in the EDX language. The data are visualised in graphic form in a way that makes them easy to understand for antenna engineers and following the common practices of this discipline.

The Validation tool for EDX implementations performs the validation of I/O functions providing access to data stored in the EDX language, providing a detailed report of all non-compliances found in the files generated by them. It is accompanied by a set of reference data files in the EDX formats to check input capabilities of other tools.

The CAD data import filter is intended to be a simple tool able to translate geometrical descriptions found in CAD files and directly compatible with a Geometry and Materials Data Dictionary into an EML file. Some advanced capabilities, like geometry healing and a smart visualisation of 3D shapes may also be included.

4. Data Dictionaries

The main purpose of a data dictionary is to define the information content of a data set. Typically the information relates directly to the values of a physical quantity and the objective of the data dictionary is to ensure that the meaning of all values is clearly and univocally defined. Most of the physical quantities involved in antenna engineering have a rather complicated structure, which inevitably inherited by their mathematical and numerical representations. The data dictionary must specify the complete structure and provide accessory information to make data exchange easy.

For example, the electromagnetic field is usually represent by engineers has an **E** and an **H** component, each being 3D vector quantities. Then depending on the type of representation, time or space domain, each vector component is a real or a complex quantity. Further to this, to fully qualify the field, say in the spectral domain, it is necessary to specify the frequency at which it is given, the notation used for the time dependency, which type of projection has been chosen for the vector components and, last but not least, the sampling grid. All these aspects are clearly defined by the Field Data Dictionary [12]. This field example will be used throughout this section to illustrate the various aspects involved in the definition of data dictionaries.

The ESA report “System Analysis and Requirements for an Electromagnetic Data Exchange Standard” [7] contains a lot more details and examples on the Data Dictionary theme. Section III “Intender readership and usage” gives an indication of where to find the information most relevant to the building of data dictionaries. A short description of a possible building procedure is found in Part 3, Section 5.

4.1 The structure of physical quantities

The large majority of physical quantities used in antenna engineering have the mathematical structure of n-D tensor field defined over some domain. In most cases the n-D tensor is mono dimensional, i.e. the quantity is a vector field. They are usually parameterised by some additional quantity, like the frequency for spectral domain representation of fields. In the end, they can be seen as n-valued functions defined over p-dimensional domains. To be given a numerical representation, these quantise once they have been discretised in some way.

Therefore it is easy and natural to give them a matrix representation. The only very significant exception to this rule is the physical description of antennas, geometry and material properties, which requires specialised representations. However, the single matrix holding the samples of the quantity does not provide a complete description. To fully describe a continuous physical quantity, no matter how it is discretised, it is necessary to fully specify the “rules” used to obtain the discrete version. There are several possible ways: sampling, piecewise or entire domain approximation, modal expansion; in all cases additional information is required to understand the meaning of the values available for the quantity itself. Mathematically this amount to specify the basis of the function space to which the quantity belongs used to obtain the series expansions from which the discrete version is finally obtained by truncation.

For instance if sampling is used there is need to provide the sample coordinates and the related reference system. Mathematically the initial quantity is a dependent variable, as its values are a function of the others, while the sampling coordinates are a set of independent variables defining the support on which the dependent one is defined. All these need to be packaged in a single whole to convey the complete meaning of data.

Whenever a physical quantity is multi-dimensional there is also need to specify how each element of the corresponding tensor is to be interpreted. For example, in a 3D vector the first element could be the x component of a Cartesian projection or the ρ component of a spherical projection or anything else. Therefore one more quantity is needed. The rule adopted in EDX is that this quantity is a string vector holding the names of the axes on which the quantity is projected, e.g. {"x", "y", "z"}. The resulting overall logical structure is depicted in figure 4.1.

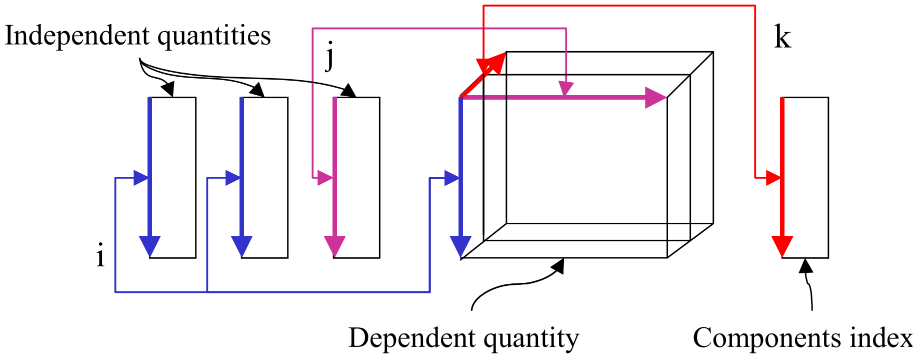


Figure 4.1 – Logical structure of a multidimensional quantity

Containers are used in EDX to keep all these quantities. In many cases, e.g. for spatial coordinates, some of the independent ones are related to a single physical entity, e.g. 3D space, therefore other containers may be used on lower levels to collect them reflecting the logical relations among data items.

Using the approach described so far, all quantities belonging to a data set can be kept jointly together in a hierarchical structure. Beyond making sure that logical relations are maintained and put in evidence , thus making data easier to understand, such approach has a another fundamental advantage: it offers a great deal of flexibility. Imagine that in a dictionary for electromagnetic fields it is chosen initially to use space sampling in Cartesian coordinates and that the three quantities X, Y and Z are used directly in the field container (figure 4.2).

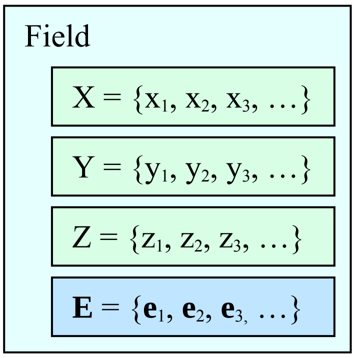

Field
$X = \{x_1, x_2, x_3, \ldots\}$
$Y = \{y_1, y_2, y_3, \ldots\}$
$Z = \{z_1, z_2, z_3, \ldots\}$
$\mathbf{E} = \{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \ldots\}$

Figure 4.2 – Field data container with fixed sampling axes

Then to use a spherical reference system it would be necessary to define a new field container, despite the fact that only the names of the three spatial axes changed. Of course the difference in meaning is quite substantial, but in the EML files the change is minimal, so a single function could handle both cases with no problems. There is more, assume now that having moved to a spherical reference, it is desired to handle the far-field case in which the ρ component can be dropped. If a Coordinates container has been used then the overall structure is maintained intact and I/O functions can easily be made that handle both cases (figure 4.3).

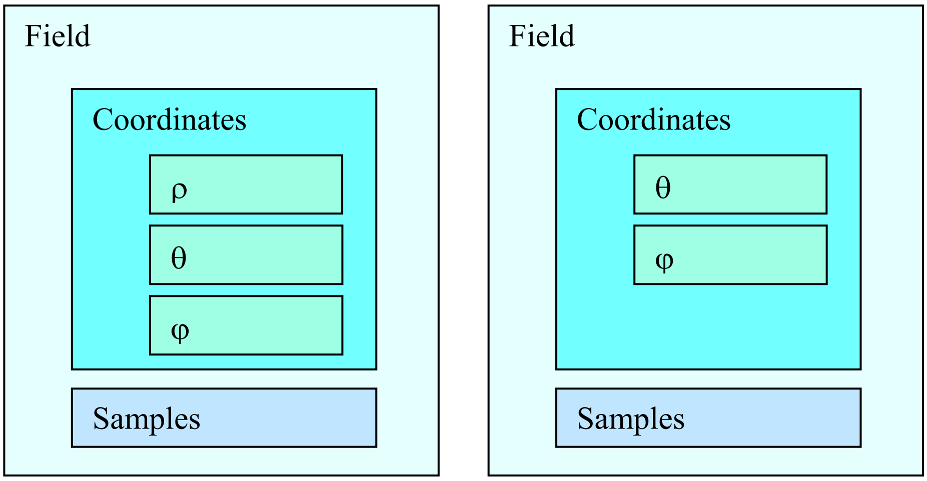


Figure 4.3 – Field data container with flexible sampling axes (via sub-container)

4.2 The structure of data dictionaries

In general, the definition of a dependent physical quantity can be split in the following three parts.

- Mathematical support: the description of the domain of definition and its sampling
- Quantity: the description of the structure of the quantity itself, e.g. a 3D vector
- Representation conventions: pinpointing all details about accessory aspects, like the implied time dependency.

EDX data dictionaries follow the same logical structure (figure 4.4).

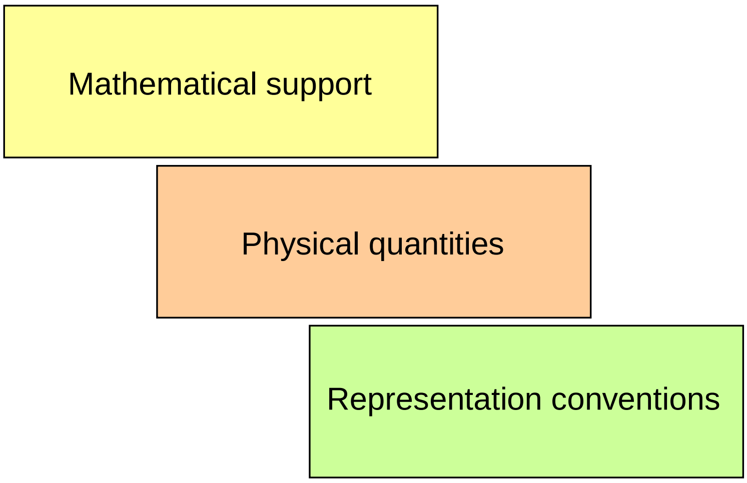


Figure 4.4 – Logical structure of a data dictionary

4.2.1 Mathematical support

The mathematical support of a physical quantity can be very simple, e.g. the real line, or complicated, e.g. a multiply-connected 5-D manifold. The data set specification must describe the numerical representation used for it, which can range from a regular sampling of a real interval to a structured volumetric mesh.

The support can in general be decomposed in a number of elements, like frequency or position. The values assumed by the quantity depend on each of these elements, which can be seen as the axes of a p-dimensional representation or as the indices of the p-D matrix associated to it. The dictionary contains the information required to describe each of them and they are called *independent quantities* or *independent variables*. In the electromagnetic field example there are three independent quantities.

1. Frequency
2. Time
3. Spatial or spectral coordinates (depending on representation) amounting to 2 or 3 independent axes

Usually only the frequency or the time dependency will matter, but in principle antennas with embedded signal-processing capabilities have a time-dependent spectral-domain field representation and therefore both dependencies have been included in the dictionary.

Sometimes the mathematical support of the physical quantity is defined indirectly. For instance when a functional expansion is used, like the spherical wave expansion for fields that implies an underlying spherical reference system. In this case the dictionary must specify the support used for the specific representation, e.g. the modal indices of the spherical wave expansion, since they constitute the actual “sampling” on which the numerical values of the modal field amplitudes are given.

In other cases it may be necessary to include information about both levels: the physical support and the mathematical support. This occurs, for instance, for meshed geometries, where both the mesh itself and the list of mesh elements used as reference to build the basis functions (e.g. the internal edges) are needed to provide a complete description of the support; the mesh to specify the actual shape of the support, i.e. the physical object, and the list of mesh elements, to specify where each currents value is located. Note also for a same mesh currents values could be associated to patches, edges (the usual RWG basis) or vertices and this needs to be indicated as well.

4.2.2 Physical quantities

The physical quantities generally have a simpler structure than their support. They are mostly scalars or vectors or tensors. Therefore in the last two cases it is only required to specify the projection selected for them. Then of course, it is necessary to specify their numerical nature (integer, real or complex). In the electromagnetic fields example and just considering the far field case, in which only two components are relevant, there is a long list, including Cartesian components, spherical components, Ludwig projections, and several others. The selection between real and complex representation is clearly linked to the selection of time or spectral domain. Today complex numbers are always represented in real and imaginary part, but it is foreseen in the future to offer the option for Euler (polar) notation.

In some cases it is useful to separate the components of a single physical quantity to make the data set more flexible. In the Filed Data Dictionary the **E** and **H** components are considered two separate quantities, since in most cases only the **E** field is used for antenna problems. However the dictionary clearly indicates that whenever they are both defined within a same data set they must share the same support, to ensure physical consistency.

Often, to save computation time and memory space, not all components of a physical quantify are specified in practice, exploiting symmetries and other physical properties. A typical case if the dropping of the vanishing radial field components in the antenna far field, another good example is the use of only a few elements of the dyadic Green’s function. The data dictionary should identify the needs in this area, since this is important to identify the most general way to represent them. The most effective way of representing the quantity depends on its structure and on the possible existence of multiple representations. Some example will clarify this rather tricky point.

Example 4.1 - *Electromagnetic field*. The **H** component is often left unspecified, so **E** and **H** are handled separately; furthermore the radial component may be vanishing (far field), therefore both quantities may have two or three components. The latter aspect is taken care by selecting among the different specifications for the vector projection included in the dictionary.

Example 4.2 - *Spherical Wave Expansion*. Out of the full 2N+1 by N space generated by the Cartesian product of the radial and azimuthal indices only about half of the combinations make sense, since it must be $n>0$ and $|m|\leq n$. However to simplify handling and to avoid inserting in the data dictionary an *a-priori* choice of the ordering of indices, it was decided to store the full rectangular matrix, waiting for the foreseen possibility to handle sparse matrices directly in EML.

Example 4.3 - *S-matrix of a reciprocal device*. Only the upper or lower triangular matrix is needed and an explicit listing of the index pairs can be used to obtain a crude but effective sparse matrix representation; in this way the S-parameter data are indexed by the pairs and not by the port indices. This case also applies to modal spectra for waveguides and to Green’s functions.

As already discussed, both physical quantities and mathematical supports have often multiple representations. For instance, the electromagnetic field can be sampled in a spherical or Cartesian reference system and others as well and it can be decomposed according to several projection systems, Cartesian, Spherical, Ludwig 1, 2 and 3, and so on. The data dictionary should include all these representations, even if initially only a few will be used. This is very important to ensure that a suitable data set structure is chosen since the beginning.

4.2.3 Representation conventions

Representation conventions include all the accessory information required to fully qualify the physical quantity. For instance, the implied time dependency convention adopted, the normalisation impedance chosen for S-parameters or the power normalisation and the phase reference point for a radiated field. Also information about functional expansions may be required, like domain of convergence, range of validity (e.g. the minimum sphere of a Spherical Wave Expansion or the far-field distance for a far field). In general, in defining a data dictionary it is important to explicitly include all assumptions and conventions used to arrive at numerical values given for the physical quantity and they should all be part in the data set, so as to make sure than reading the corresponding EML file there can be no doubts about the meaning f its content.

In several cases it is possible to define reference conditions, which could be assumed to hold unless differently specified in the data. Then the data dictionary could report this assumption and avoid repeating it in each and every data set. For instance, it is customary to assume a normalisation impedance of 50Ω for S parameters. However, such approach should be used very sparingly, to void the need to always go back to a “reference book” to fully understand the data. A data set should contain all information required to fully qualify a physical quantity, with the exception of those cases in which the chance of ambiguity or doubts is so small that the user of data will seldom need to go back to the Data Dictionary definition to understand them.

4.3 A data dictionary example

An example of a data dictionary should help to clarify this rather thorny subject. Again the electromagnetic field is taken as reference; in particular the field radiated by a source in free-space, which is a relatively simple entity. It is constituted by a pair of time-varying 3D vector fields, **E** and **H**, defined over the entire space minus the source region and it is assumed to be sampled within a region of space over a fixed grid of points.

The corresponding data dictionary will include the definition of the following quantities, called classes to underline the fact that no values are associated with them: `Field`, `Time`, `ScanRange`, `ProjectionComponents`. The root class will be a container hosting all of them into a single data element, the `ElectromagneticField`, defined as follows.

```
Class  ElectromagneticField
      Component  Time
      Component  ScanRange:*                ! * = all subclasses
      Component  ProjectionComponents
      Component  Field
```

The `Field` class will define the **E** and **H** vectors, specifying the measurement units (V/m and A/m respectively), the data type used to quantify them (real numbers) and the size of each of their samples. It will also specify the overall dimension, i.e. total number of sampling points taken in space and time.

```
Class  Field
      Dimensions (Time*ScanRange)
      Component  E
            Units      V/m
            Type       real
            Size       (ProjectionComponents)
      Component  H
            Status     optional
            Units      A/m
            Type       real
            Size       (ProjectionComponents)
```

Note that the dimensions are formally specified as the product of three other quantities, meaning the product of their actual sizes. The product is enclosed in parenthesis to indicate this indirect relation.

The `Time` quantity is much simpler, just a real vector with an undefined number of elements.

```
Class  Time
     Units      s
     Type       real
     Size       1
```

The `ScanRange` quantity that specifies the space sampling is more complicated. First the sampling can be made in many different reference systems, thus a list of alternative subclasses is included in the definition. This fact is also indicated by the `ScanRange:*` notation in the `ElectromagneticField` declaration. Each subclass specifies the names of the 3 axes, their measurement units and data type. Also in this case no dimension is specified, so each component can be a vector of any length, the actual overall dimension of the `ScanRange`, i.e. the one used to compute the actual dimensions of the `Field` quantity, is the Cartesian product of these vectors.

```
Class  ScanRange
     Subclass   Cartesian
          Component  x
               Units      m
               Type       real
          Component  y
               Units      m
               Type       real
          Component  z
               Units      m
               Type       real

     Subclass   Cylindrical
               …

     Subclass   Spherical
               …
     …
```

The `ProjectionComponents` class has a somewhat different nature. In principle, it would have been possible to state that the **E** and **H** vectors have length 3 and avoid any further complication. However the use of an extra class offers quite a number of possibilities. First it allows to specify in a convenient way the reference system chosen for the projection of the field vectors. Second it does so, using strings, which convey the exact names used in antenna engineering books, which can be used for data display and plotting. Third by defining subclasses with different sizes it would be very easy to take of the situations in which only two components are needed or known (e.g. on an equivalent surface or in the far field).

```
Class  ProjectionComponents
     Type       string
     Size       3
     Range      {[“x”,“y”,“z”];
                 [“\theta”,“\phi”,“r”];
                 [“a”,“b”,“r”];
                 [“Az”,“El”,“r”];}
```

TEX notation, e.g. `\theta` to indicate the Greek letter θ, has been chosen for the data dictionaries developed so far for name strings since it offers readability while supporting pretty printing of projection axes names, e.g. as labels in plots, with many existing tools.

It is also interesting to see how the simple field dictionary illustrated above can be easily extended to cover the very common case in which fields are given in frequency-domain (Fourier transforms) rather than in time-domain and the time dependence is replaced by frequency dependence. Often spectral field representations are also used instead of space domain ones. The following examples list all changed and new quantities required to extend the dictionary to frequency domain fields. Note in particular the addition of the `TimeDependency` class to specify the implied time dependency convention and the fact that it is considered an attribute and not a component (see section 4.4 for an explanation).

Changes

```
Class  ElectromagneticField
     Attribute  TimeDependency
     Component  RepresentationDomain
     Component  Time
     Component  Frequency
     Component  ScanRange:*
     Component  ProjectionComponents
     Component  Field

Class  Field
     Dimensions (Frequency*Time*ScanRange)
     Component  E
          Units      V/m
          Type       (RepresentationDomain.TimeType=”time” ? real :
                          complex)
          Size       (ProjectionComponents)
     Component  H
          Units      A/m
          Type       (RepresentationDomain.TimeType=”time” ? real :
                          complex)
          Size       (ProjectionComponents)
```

New quantities

```
Class  TimeDependency
     Type       string
     Size       1
     Range      “+j \omega t”


Class  RepresentationDomain
     Component  TimeType
          Type       string
          Size       1
          Range      {“time”; “frequency”}
     Component  SpaceType
          Type       string
          Size       1
          Range      {“space”; “spectrum”}

Class  Frequency
     Unit       Hz
     Type       real
     Size       1
```

The `ElectromagneticField` class is extended to hold the extra classes and, obviously, a new `Frequency` class is added. The `Field` class needs also some modification to accommodate for the fact that now the field vectors can be either real (time domain) or complex (frequency domain). The syntax used is similar to that of the C/C++ condensed if, the part before the question mark is a condition, the first token after the question mark applies if the condition is true and the one after the colon otherwise. Finally the `RepresentationDomain` class is introduced to specify all possible combination of Fourier transformations for time and space.

Clearly if the need arises at a later time, it would also be possible to specify independent transforms for each space axis just adding a subclass to the `RepresentationDomain` class.

4.4 Alternative representations of simple quantities

The EML (see section 5), following the XML approach, offers two different ways of expressing values associated to a quantity: as components and attributes. The basic difference between the two is that the second can only hold single immediate values. The more subtle difference, much more important from a data dictionary perspective, is that attributes do not offer room for extensions of the dictionary. As shown in the example in section 4.3 components may actually refer to another class, which in turns hold several components, thus leading to the possibility of very sophisticated organisations of the data. By carefully designing this data structure it is possible to obtain a very flexible dictionary open to future modifications that do not invalidate the original structure. On the contrary, since attributes just old single immediate values, their definition can not be modified or changed into a component without making the resulting EML files incompatible with the older version.

The choice whether some quantity should be represented by a component or an attribute is quite common. The rule to be applied is that only simple quantities that can be expected not to need changes in the evolution of the dictionary should be handed via attributes. To be more precise, the only thing that can be changed in an attribute is the list of admissible values. For instance in the field dictionary discussed in section 4.3 the TimeDependency attribute can be modified to extend its range from “+j\omegat” to {“+j\omegat”, “-j\omegat”}. On the other end, the RepresentationDomain could have been split in two attributes, one for the time-frequency alternative and the other for the space-spectral one, but being conceivable that it is extended to cover the case of mixed spatial-spectral representation a component is the proper choice.

4.5 More complex data structures

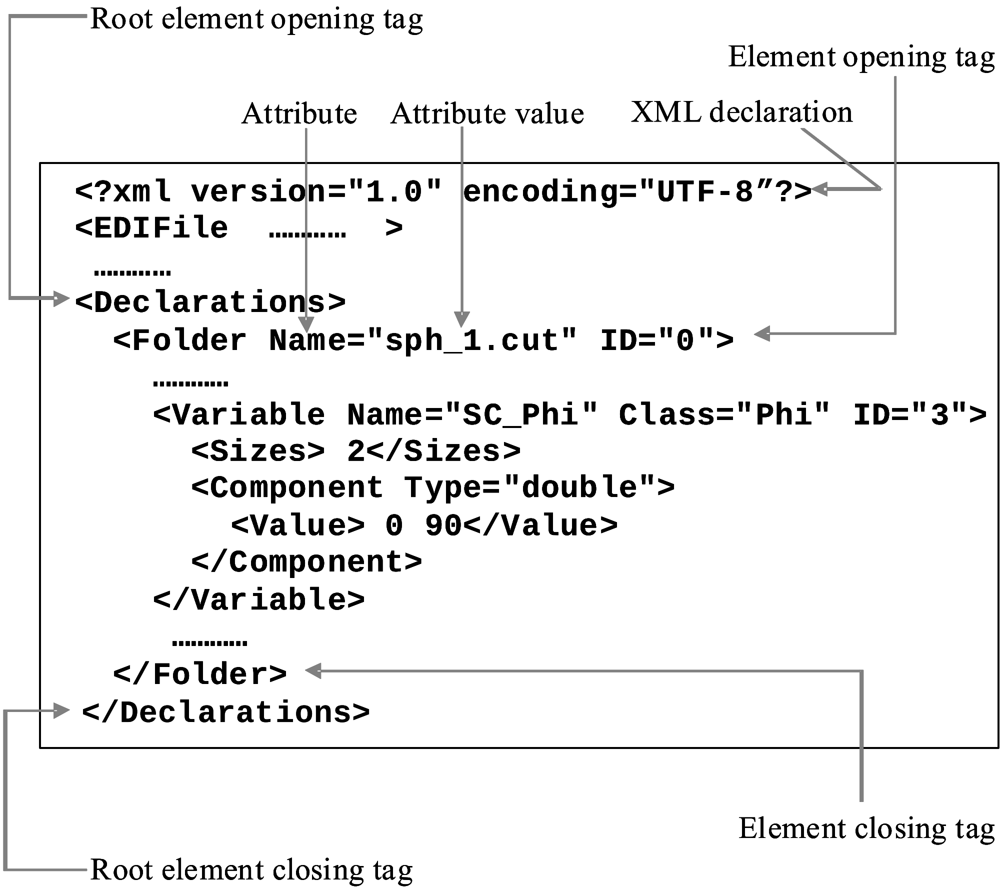
There are chunks of data having a much more complicated logical structure than those discussed so far. The most prominent example is the geometrical representation of objects. Another one is the description of an antenna, even when limiting to its main elements, i.e. not listing all screws and washers. In general complex representations require the use of data trees or networks, in order to capture all links existing among their different components. A data dictionary for such entities must capture these relations, which are usually very important. For instance, it is relevant to know the position of the feed with respect to the reflector of an antenna. The EML provides special facilities to capture this complexity. However they are not discussed in this introductory presentation to avoid overloading it. The interested reader is addressed to the EDI manuals, until the matter will be covered by further EDX manuals.

5. The Electromagnetic Mark-up Language

Initially it was believed that the data needed in EM data sets could be represented with extremely few basic blocks – simple named matrix building block (vectors and scalars being special cases of the matrix) where the elements of the matrix could be of the standard types such as integers, reals, complexes, logicals and characters plus a few extra features such as attributes. This would be little more structured version of the very common traditional data file organisation with a sequence of blocks each initiated by a tag on a separate line followed by some lines with numbers or characters. The analysis mentioned in section 3.2 and the work in the ACE and EAML groups revealed that a much more structured approach was needed, and here the XML was right at hand. The EML is thus very structured *but in contrast to what might be expected it is not very big language*.

5.1 (Absolute) XML basics

The absolute basic building blocks of the eXtendable Mark-up Language are *tags*, *elements* and *attributes* as shown in figure 4.1 where one of the EML elements are used as example:



```
<?xml version="1.0" encoding="UTF-8″?>
<EDIFile  …………  >
 …………
<Declarations>
  <Folder Name="sph_1.cut" ID="0">
    …………
    <Variable Name="SC_Phi" Class="Phi" ID="3">
      <Sizes> 2</Sizes>
      <Component Type="double">
        <Value> 0 90</Value>
      </Component>
    </Variable>
     …………
  </Folder>
</Declarations>
```

Figure 5.1 - Basic XML concepts

The key building block in XML is the *elements*. An element is some text (denoted the element body) bounded by a set of tags denoted the opening tag and the closing tag. The tags are strings initiated by '<' and ended by '>'. The opening tag contains the element type ID (e.g. 'Variable' in fig. 4.1) followed by optional *attributes*. An element attribute has a name (a string) and a value (a string in double quotes). Hence the opening tag

```
<Variable Name="SC_Phi" Class="Phi" ID="3">
```

opens a element of type *Variable* which has 3 attributes - *Name*, *Class* and *ID*.
The closing tag of an element only contains the division character '/' followed by the type ID i.e. `</Variable>` in the example above.

Very often it is adequate to use a so called *empty element*. Empty element are singleton tags (also called empty tags); they have no text between a pair of tags and the singleton is ended by the string '/>'. The information in an empty element is held in an attribute:

```
<Domain Reference="SpherCut_Phi"/>
```

A key and very important feature of XML is that *elements can be nested* to any level i.e. one element can contain other elements. In figure 4.1 the element type *Declaration* contains the element type *Folder* which again contains *Variable* etc. The nesting thus enable designer to build a rather complex language by defining simple types that can then be put into more complex types.

Until now this mini outline of XML of has only been about *form*. Basically any gibberish that satisfies the above syntax can be considered as XML but it will of course be of no use. Hence the next step is to add *meaning* and this is really where the designer starts to eXtend. The idea is to define a set of *types* which is needed to hold the data of the desired data set. A type is made in a type definition. The type has a name (e.g. *Declaration*), some attributes if needed and a body which can contain simple data or it may in itself contain elements of other homemade types. In the next section this is done for the EML which is thus an example of the eXtension process. A data set will then be a collection of instances of the defined types.

This very, very limited discussion of XML is only intended as an outline for the reader to understand the following sections. There is of course much, much more to be said about XML. The interested reader is referred to one of the many web-sites (a search for "Introduction to XML" on the web yields more that 100000 hits).

5.2 The EML types

The EML includes four main sections of a data sets (e.g. in a file). These four sections are

- the *header*
- the *declarations*
- the raw *data*
- the *application data* section.

The rest of this chapter describes in detail each one of them. The *declarations* section, which conveys the meaning and defines the structure of data is the core of an EML and therefore has received most of the attention in the discussion.

5.2.1 The header

With today standards people is accustomed (or even expects) that any data set is followed by a practically complete amount of information about its origin and the standard to which it complies. In EML this information is collected in the *<Header>*. This element is mandatory and it looks as follows:

```
<Header>
  <Stamps>
    <Version>EDI Version 1.00.00</Version>
    <Format>XML</Format>
    <DateTime>2008-01-20T13:27:53Z</DateTime>
  </Stamps>
  <Origin>
    <Tool>
      <Name>GRASP9</Name>
      <Version>9.4.01</Version>
    </Tool>
    <Project></Project>
    <User>
      <Name>Poul Erik Frandsen</Name>
      <Affiliation>TICRA</Affiliation>
    </User>
  </Origin>
  <UserText>Field data in cuts</UserText>
</Header>
```

The <Header> can contain three elements: <Stamps>, <Origin> and <UserText>.

5.2.1.1 <Stamps>

The <Stamps> element contains three elements that will be added automatically by the EDI.

The <Version> tells which version of the EDI that has created the file. It is here assumed, that the file has been created with functions in the EDI library. Some developers may find it attractive to manipulate an EDI file with homemade functions (routines) that perform all input and output using READ and WRITE (Fortran90) or getf and putf (C/C++) statements. *This is definitely not recommended!* Rather, all access should be made with the EDI library.

Next, the <Format> tells that this is an XML file. When the EDI is extended with features for other formats (e.g. HDF5), a single format switch will be available for the developer, and a very simple extension of a software tool that uses the EDI will enable output in any of the available formats.
The <DateTime> tells the time when the file was created i.e. when it was closed.

5.2.1.2 <Origin>

In this element information is collected about

- the software tool that created the file
- and in which project
- and by which user

The <Origin> in itself and all contained elements are optional. It is, however, polite if a tool and its user reveal this information. The high level EDI Library routines for output of complete fields or currents (known as level 2 routines) will, however, leave these fields empty.

5.2.1.3 <UserText>

This element is intended for any number of text lines that the output tool or the user might find adequate. In the case of a parameter study where the same tool is run many times one may add information about which parameter configuration was used to create the present data set. The element is optional.

5.2.2 The Declarations and the EML <Variable>

The "real" data i.e. all the various numbers and strings that describe physical and mathematical entities in an electromagnetic data set such as a set of currents are managed in the two elements <Declarations> and <Data>. Each data item is described in a <*Variable*> element in <Declarations> whereas the corresponding numbers or characters (the raw data) appear in the <Data> element (see section 5.2.3). This is the rule unless few values are involved, in which case the data are kept together with the declaration in a local <Value> element. The reason for this separation is simple. By keeping all data descriptions in a fairly short element (usually in the top of the file if the data set is stored as such), it is simple for a user to get a clear overview of the content of the data set e.g. using a standard text editor. Further, the software that handles the data set can quickly read the short descriptions and perform practically all consistency checks before massive input of large amounts of numbers is initiated. The receiving software tool can actually quickly check if the content of the data set is relevant or if it should be rejected.

The <Declarations> element looks as follows:

```
<Declarations>
   <Folder Name="folder_name" ID="0">
      <Variable …>
         …
      </Variable>
         …
      <Variable …>
         …
      </Variable>
   </Folder>
</Declarations>
```

At the present there can only be one <*Folder*> with the integer ID equal to 0. This element type is intended for future use and will not be discussed further here. If the data set is saved in a file, the EDI Library will set the folder name equal to the file name.

As discussed in section 3.2, the EML must be able to represent two fundamental and very common data structures: n-dimensional matrices, to represent nD-tensor fields, and hierarchical compounds, to represent the logical and lexical connection among data. The key element type designed for this purpose is the <*Variable*>. The type name is inspired by

NetCDF and it designates that the content may be changed if e.g. a data set is input, modified and later output from a program.

5.2.2.1 <Variable> with data

The <Variable> appears in two forms: 1)

1) 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.

2) with references to other variables. The variable will not have any real data (with the exception of simple EML Attributes – see below). Rather, it will contain *references* that points at other EML variables in the same file.

In the first case, an instance looks as follows:

```
<Variable Name="variable_name" Class="class_name" ID="int>0">
   <Attribute Name="attrb._name1">attrb._value1</Attribute>
   ……
   <Attribute Name="attrb._name_j">attrb._value_j</Attribute>
   <Sizes> list of n positive integers (n > -1) </Sizes>
   <Domain Reference="first name of another variable"/>
   ……
   <Domain Reference="n’th name of another variable"/>
   <Component Type="a valid EDI type">
      <Value> some values of the above type </Value>
   </Component>
</Variable>
```

The italic are where the actual data appear. The <Variable> type has three XML attributes: Name, Class and ID. The *name* must be unique within one folder and the usual rules for names apply: first character must be a letter, then a sequence of letters, digits or underscore (no special characters and no blanks). These naming rules apply to all names in EML.
The class name is optional, but it *is strongly recommended that classes are used* for all variables in a data set. Not doing so will basically prevent the use of many existing and future features of tools based on EDX. Hence all instances of variables should belong to some class from the joint pool of classes derived from the Data Dictionaries. If no class is available, the developer should prepare a new class together with the other developers that shall use the data set. The class description specifies exactly what to find in the variable i.e. between the element tags (see sections 3.1 and 4).
Finally, the ID is an internal integer identifier managed by EDI – don’t mess with these in an editor.

The first content of the <Variable> type is a sequence of EML attributes (*don’t confuse these with the XML attributes that appear in the opening tag*). The EML attribute is a type that has been designed by the ACE-EAML group to hold simple descriptive data which in general do not change. The attribute has a name. As an example, the time dependency of a field is stored in an EML attribute:

```
<Attribute Name="TimeDependency">+j \omega t</Attribute>
```

Next you will see the <Sizes> element. This element is empty or it contains a sequence of integers. If it is empty, the <Variable> either holds a simple *scalar* of one of the EDI types or the variable is a container i.e. it has no data in itself but references to other variables (see 5.2.2.3 4.2.2.2). A scalar <Variable> will look as follows:

```
<Variable Name="My_Control1" Class="Switch:String" ID="3">
  <Sizes></Sizes>
  <Component Type="string">
    <Value>On</Value>
  </Component>
</Variable>
```

If the <Sizes> is non-empty, the integers tells the size of an n-dimensional array also denoted a *Rank* n array where each element is a <*Component*> of one of the available EDI-types. S*tructures* as components are under consideration. If the <Variable> has only one <Component> there is no need for naming it. If, however, more components are needed, each component has a name:

```
  <Component Name="Double_Components" Type="double">
    <Value> 1 2 3</Value>
  </Component>
```

In the following example a 1D variable of class Frequency has a component of EDI type double i.e. a double vector of length 3 due to the <Sizes>. (Since no units are specified, the default unit Hz is assumed. See more about defining and using a class in section 4).

```
<Variable Name="My_Frequency" Class="Frequency" ID="1">
  <Sizes> 3</Sizes>
  <Component Type="double">
    <Value> 300000000 310000000 320000000</Value>
  </Component>
</Variable>
```

The rank can be any positive integer. In the Fields DD a rank 7 array may occur i.e. there will be 7 integers in <Sizes>. A <Variable> can have any (positive) number of <Components> of different EDI types, but they are all bound to the same <Sizes> as in the following example, where there are 3 integers in the first component and 3 strings in the second:

```
<Variable Name="My_Mixed_Var" Class="Mixed_Class" ID="1">
  <Sizes>3</Sizes>
  <Component Name="Double_Components" Type="double">
    <Value> 1 2 3</Value>
  </Component>
  <Component Name="String_Component" Type="string">
    <Value> "String1" "String2" "String3"</Value>
  </Component>
</Variable>
```

In the above example three values appear in a <Value> element inside the <Variable>. This option can be used in EML if the number of values is small. If the number of values is larger, the values will appear in the <Data> element to avoid long <Variable> elements in the <Declarations>. If the file has been created with the EDI, the library will determine when the values shall appear in the <Variable> and when in <Data>.

5.2.2.2 Domains and dependencies

The last element type that appears in a <Variable> with own data is the <Domain Reference="…"> element which contains a reference to *another EML variable* in the file.. This element - an empty XML element (see section 5.1) - is used to tell that the referenced variable is part of the domain of the <Variable> it self.

EML <Variables> that have <Domain>'s are called dependent. Otherwise they are called independent – such as the Frequency variable shown above. The Reference attribute points at another EML variable that must be in the data set (e.g. in the file). In the following example there are four <Domain Reference='…'/> specified telling that the values of the pattern was calculated at points obtained as the Cartesian product of the four variables:

```
<Variable Name="My_Ptn" Class="Field:RadiationPattern" ID="6">
  <Sizes> 2 2 161 1</Sizes>
  <Domain Reference="My_Ptn_ProjectionComponents:Ludwig3"/>
  <Domain Reference="My_Ptn_Phi"/>
  <Domain Reference="My_Ptn_Theta"/>
  <Domain Reference="My_Ptn_Frequency"/>
  <Component Type="dcomplex"/>
</Variable>
```

Clearly, the <Sizes> specification of the dependent <Variable> must correspond to the <Sizes> of the <Variable> on which it depends. Unfortunately, the EDI Library will not and can't perform this check! It is thus up to the developer to ensure that the <Sizes> are consistent when a tool outputs a data set. (It is quite easy to implement the check for a <Variable> which has been input using EDI function calls, but then what shall be done if the <Sizes> doesn't fit?).

The <Domain> can thus be used, and should be used, to tell a receiving software tool the dependencies among the various elements in the data set. Finally, in the <Sizes> in the above example one can see that "My_Ptn" is a 4D double complex array with 1×161×2 ×2 elements and obviously corresponding to 1 frequency, 161 θ-values, 2 φ-values and with two field components. Note, that the EDI Library stores the domains in reverse order of the one known within the software tools (the order in which the domain variable names are given to an EDI function). So basically the double complex field pattern values have been created in a loop structure as follows:

```
Loop over Frequencies (here 1)
  Loop over Theta values (here 161)
    Loop over Phi values (here 2)
      Calculate component 1 and component 2
    pooL
  pooL
pooL
```

In practice the developer/user should avoid to mess around the actual values in a file and they are often not saved in nice columns, but in practice it is often needed to fetch the data with a tool that doesn't "speak EDX" and in that case the above information is vital.

5.2.2.3 <Variable> with references

In 5.2.2.1 the <Variable> with own data was introduced. For several reasons, however, another option has been added to the EML <Variable>. Here the <Variable> doesn't posses its own data. Rather, it has references to one or several other EML <Variable>'s. A typical example:

```
<Variable Name="My_Ptn_Field:Far" Class="Field:Far" ID="7">
  <Attribute Name="SpaceTypeAxis">Space</Attribute>
  <Attribute Name="TimeDependency">+j \omega t</Attribute>
  <Attribute Name="TimeTypeAxis">Frequency</Attribute>
  <Sizes></Sizes>
  <Component Reference="My_Ptn_Frequency"/>
  <Component Reference="My_Ptn_ScanRange:ThetaPhi"/>
  <Component Reference="My_Ptn_ProjectionComponents:Ludwig3"/>
  <Component Reference="My_Ptn_Field:RadiationPattern"/>
</Variable>
```

In the above example the applications of <Component Reference="…"> is shown where the <Variable> element now appears as a compound, which tells the connection among data. The compound is also denoted a container. Note that the <Sizes> element is empty – it must however be there. Since compounds can reference other compounds (obviously a ring of references is not allowed), this type of <Variable> enables a joint handling of very complex data sets – all <Variables> in the data set can be connected either with a <Component Reference="…"> or with a <Domain Reference="…"> in a multi-dimensional treelike structure and thus show the logical connection among data. If a compound is not referenced by any other <Variable> it is a top-level compound, i.e. it acts as the entry to a large data set.

The compound also enables the handling of several data sets in the same EDI file, and some data sets may even share some features. Two fields may e.g. reference the same Frequency variable. The use of compound is not mandatory. A group of developers can decide to represent their data set with a set of unconnected <Variable>s. The handling of the references is, however, quite easy when the idea has been caught and the EDI Library has a nice collection of functions for managing references.

5.2.2.4 Units

As these notes are being prepared, the EML is under further development. The observant reader may have noticed that the issue of units has not been discussed at all. Initially the idea was that default units should be agreed upon for all elements in a data dictionary and units should thus be implicit. As an example it was agreed that the unit for Frequency should be Hz (thus usually leading to very large numbers in <Variable>s of this class).
The first experiences quickly revealed that this was much too stringent. For this reason a new Unit attribute has been added to the <Component> element which now may look as follows:

```
  <Component Name="…" Unit="…" Type="(some EDI type)">
```

The Unit is optional. If it doesn't appear, the data item is given in default units as agreed upon in the Data Dictionary. However, it strongly recommended to specify units since their availability makes data much safer to use, e.g. allowing display of units in processing and visualisation tools.

5.2.3 The <Data> element

If the number of data values in an EML <Variable> is larger than a few, the values will not appear together with the declaration in a <Value> element. Rather, the data will appear in a corresponding <Variable> in the <Data> element with exactly the same name. The relation between the <Variable> in <Data> and the corresponding <Variable> element in the <Declarations> is simple: They have the same name, the components have the same names and EDI types and the attribute RefID has the same value as the ID attribute in the declaration.

```
<Variable Name="(declared variable name)" RefID="(integer>0)">
  <Component Name="(component name)" Type="(EDI type)">
    <Value>
      …… at lot of data e.g. hundreds of numbers ……
      ……
    </Value>
   <Component>
   ……
   <Component Name="(component name)" Type="(EDI type)">
    <Value>
      …… at lot of data e.g. hundreds of numbers ……
      ……
    </Value>
   <Component>
</Variable>
```

In a file the raw data are stored in a way which is adequate for the EDI Library functions (usually 2 numbers pr. line). There is, however, no rule in the EML for this format. The *sequence* of the data values are, however, related to the declared <Domain Reference="…"> elements of the <Variable> variable in question.

5.2.4 The <Application_data> element

The last type of element in the EML is the <ApplicationData>. The syntax is extremely simple:

```
<ApplicationData>
    … any data what so ever …
</ApplicationData>
```

This section is intended for a *programs private data* i.e. data that are not part of the data set in question. As an example the field data dictionary doesn't 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. The form of data items inside the element is not prescribed, but it is of course recommended to use an XML extension or even use an EML-like form if possible. This will make it easier to maintain the content in the future and a user can easily understand the data if the file is inspected with an editor. Hence in stead of using

```
<ApplicationData>
    0.04
</ApplicationData>
```

one should use something like

```
<ApplicationData>
   <Feed_position Unit=”mm”> 0.04 </Feed_position>
</ApplicationData>
```

or maybe even

```
<ApplicationData>
   <Variable Name="Feed_position">
      <Component Unit="mm" Type="double">
         <Value> 0.04 </Value>
      </Component>
</ApplicationData>
```

The latter alternative does use much, much more space in a file - but then it is infinitely more informative.

5.2.5 Organising a data set with EML

The few and quite simple EML features discussed in section 5 can be used to organise rather complex data sets in an EDI file. In a data set, data are related. As an example, a far field will consist of many variables each of which describes various features of the field such as sampling, frequencies, polarisation, the actual E and H field values etc. In the file all these variables reside as Lego blocks. *How can they be kept together such that the whole field appears as a unity to a tool that reads the file?* The observant reader will know the answer: using component and domain references.

The design and organisation of the elements of the data set is determined by the data dictionary. When it has been determined which data are kept as attributes and variables, the easy solution is to make containers which link the data together. There will always be one *main (root) container* variable that will be the *entry* to the data set giving component references to many other variables. The class of this container will be the agreed upon class of the data set e.g. *Field:Far*. The container will have some mandatory component references and some optional – the later will only be used by some tools. This will all be laid down in the data dictionary. Other containers may then be needed on a second level to organise other groups of variables. The principle is shown in figure 5.2.

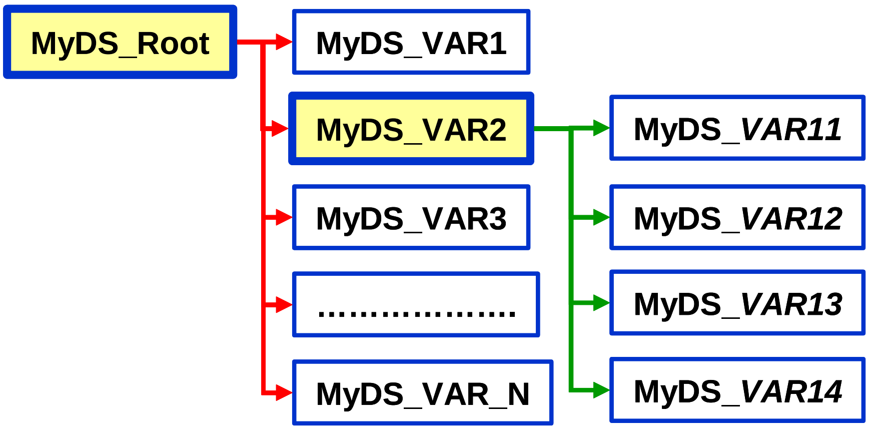


Figure 5.2 - Organisation of a data set using containers (MyDS_Root and MyDS_VAR2)

Besides the component references there will typically also be a need for domain references which at the same time link variables together and tells which variables depends on which. Finally, we note that different data sets may share various features e.g. a set of currents and a near field may reference the same frequency variable.

5.2.6 Advanced data structuring

As already mentioned in Section 4.5, EML offers the elements to build data sets that are much more complex than those discussed so far. In particular, it offers the possibility to assign dimensions to components, i.e. to state that each element of a component is constituted by an n-dimensional matrix of elements. This feature makes it easy to store vector and tensor quantities. Furthermore components can be nested, i.e. a component can be a container for other components of lower level. In this way it easy to build tree-like data structures, similar to the derived types available in many programming languages (e.g. `struct` in C/C++, `TYPE` in FORTRAN90, `record` in Pascal). Links among these elements can be created in various ways so as to capture the full complexity of tree- and network-like data structures.

These advanced topics are not discussed in this introductory presentation to avoid overloading it. The interested reader is addressed to the EDI manuals, until the matter will be covered by further EDX manuals.

5.2.7 EML syntax in Extended Backus-Naur Form

Here we will insert the real EBNF for EML later.

```
{<Folder Name="…" ID=(integer>=0)1>
     {<Variable Name="…" [Class="…"] ID=(integer>0)>
          {<Attribute Name="…">string</Attribute> }*
          <Sizes>[comma separated list of integers>0]</Sizes>
          (
               { <Component Reference="…"/> }*
          |
               [{<Domain Reference="…"/> }*]
               {<Component Name="…" Type="…">
                         [<Value>…</Value>]
                </Component>2}+
          )
     </Variable> }*
</Folder> }+
```

---

[1] ID numbers are unique across the entire Declarations section. The first folder has ID=0 and therefore ID of variables will start at 1.

[2] ID numbers are unique across the entire Declarations section. The first folder has ID=0 and therefore ID of variables will start at 1.

6. Electromagnetic Data Interface

6.1 Outline of the EDI

A file written using the EML language is quite simple to understand, still it would take a lot of effort to program the functions to read and write it. Clearly a library including the functions required for a rapid and easy creations and manipulation of EML data files offers many advantages to antenna engineers, researchers and EM software developers. Further, once an interface is agreed, the underlying implementation can be extended, updated and maintained without changes in the software that utilizes the library. The Library developed for this purpose is the Electromagnetic Data Interface (EDI) which can be called from software implemented in C++, Fortran90 and Matlab running on various Windows and Linux platforms [14,15,16,17].

EDI is composed of three layers offering increasingly higher levels of functionality. The first level, *EDI Toolkit*, offers all the basic functionality required to access XML files compliant to the EML grammar described in section 5. EDI Toolkit functions are the library's private i.e. they can't be used by a developer. This level will thus not be discussed further here.
EDI Level 0 offers a set of functions to open and close an EDI file. Once opened, there are functions to read, write and query EML <Variables>, element by element, and to access the information contained in the Header and Application Data sections. *EDI Level 1* includes functions for data manipulation such as data selection (slicing, sorting, extracting), permutation of matrix indices and incremental variable manipulation, including update, insert, overwrite, append and resize.

The library is designed to allow the use of other low-level formats besides EML. In particular, high-performance standards, like NetCDF and HDF5, are expected to become available in the future, without having to change the implementation of tools using EDX i.e. the interface to EDI functions will not be changed.

The services provided by EDI Level 0 and 1 are accessed via an application programming interface consisting of about 100 functions. Most of them perform the same kind of operation (read, write, and query) on different types of numerical data (integer, real, complex…), therefore the number of truly different operations is much smaller. Yet, EDI is not really immediate to master. Properly written access functions to handle a complete data set require quite a number of function calls and checks which are relevant to the relations among different variables and need to be programmed by the user.

To simplify this task, *EDI Level 2* offers a very simple way to access complete data sets, e.g. all data associated with a Field can be read or written with a single call. The price to be paid for such extreme simplicity is stiffness; data can only be accessed following a predetermined pattern. Two more alternatives have been devised to offer increased flexibility at the price of a little more complexity: the EDX Handler and the EDX Translator. They are very different from each other and from EDI Level 2, but they both rest on the Level 0 and 1 functions.

The present document is written for the EM engineer or scientist whit a reasonable programming experience and therefore all 3 levels will be discussed here.

Finally, it must be noted, that the present EDI doesn't include any features for channels IO. The EDI library manages data contained in disk files and this is what will be assumed for the rest of this document. Second, there is only room to introduce the most important features. The reader is referred to the EDI users manuals [14,15,16,17] for a complete discussion of the library.

The present section shall *not repeat the users manuals*, which are quite thorough. In stead the main functions are presented. Details will only be discussed when adequate as a supplement to the manuals. Since there is only a Fortran API for the EDI Levels 0 and 1 the following sections will be based on this API.

The EDI Fortran API functions are organised as follows:

- File functions
- Header functions
- Declaration functions
- Data IO functions
    - Normal put and get functions
    - Direct access functions
    - Data IO with Domain reordering
    - Advanced Level 1 functions
- Application data access

Clearly a <Variable> must be declared before it can be manipulated. Otherwise, the four main sections can be accessed in any order that may fit a specific tool because the EDI will store the information in its own data structures.

6.2 EDI Level 0: Basic functions.

6.2.1 File functions

The three functions for managing the files are for *opening*, *closing* and asking if an EDI file *exists*. In Fortran77/90 the functions are called

- EDI_FILE_OPEN,
- EDI_FILE_CLOSE and
- EDI_FILE_EXISTS.

In the users manual you will note that the actual file name (or path) is only used as input when the file is *opened* or if an *existence check* is made. As soon a the file is open, all other function have an opaque integer "EDI file identifier" (`file_id`) argument which indicates the EDI file to use.

The open function has a status argument 'status' which must be equal to 'new', 'old' or 'unknown'. The latter input can be used when an EDI shall be opened without knowing if it already exists. The EDI_FILE_EXISTS will check both if the file exists and *if it is an EDI file*.

6.2.2 The header functions

EDI is equipped with a complete set of functions for manipulating specially all the elements in the header (see section 5.2.1) i.e. relevant put and query the values of these elements:

- EDI_HEAD_QUERY_STAMP
- EDI_HEAD_PUT_STAMP_FORMAT
- EDI_HEAD_QUERY_ORIG_TOOL
- EDI_HEAD_PUT_ORIG_TOOL
- EDI_HEAD_QUERY_ORIG_INFO
- EDI_HEAD_PUT_ORIG_INFO
- EDI_HEAD_QUERY_TEXT
- EDI_HEAD_PUT_TEXT
- EDI_HEAD_REMOVE_TEXT

The use of these functions is straight forward with one exception: The function EDI_HEAD_PUT_TEXT has an array of strings as input which are the strings to put in the <UserText> element in the header. It also has a logical (Boolean) input array argument named `text_type` of the same length as the string array which is a little peculiar. It determines whether each string for the <UserText> element shall be appended in a <Line> element or "as is". Hence if string(1) = 'Mytext1', string(2) = 'Mytext2' and string(3) = 'Mytext3' and if text_type(1) =.FALSE., text_type(1) =.TRUE. and text_type(1) =.FALSE., then the <UserText> element in the file appears as:

```
<UserText>Mytext1<Line>Mytext2</Line>Mytext3</UserText>
```

i.e. 'Mystext2' is in a <Line> element.

6.2.3 Managing <Variable>s in <Declarations>.

Besides the <Header>, the rest of the file will contain a set of <Variables> that hold the actual data (plus optionally an <Application> element). Before any <Variables> can be used they must exist. If the data set has been input from a file, there will already be a set of <Variable>s which the program will want to know all about. If, on the other hand, the data is going to be saved in a file, the necessary <Variable>s must be created and organised. The EDI has an extensive set of functions for such managing of the <Declaration> element. Basically there are three main groups of functions. 1) *Output*: Create <Variable>s with various features for a data set which is going to be saved in a file or 2) *Input*: When an EDI file has been opened these functions can be used to inspect (query) the <Variable>s in the file. 3) Besides these two important groups, some functions are made for *updating* a set of <Variable>s that has either been input or created.

The complete list of functions is listed in tables 6.1, 6.2 and 6.3.

Table 6.1

Functions used at INPUT	***Purpose***
`EDI_VAR_QUERY`	Asks if an EDI variable exists and gets information about it - *not* including domains
`EDI_VAR_QUERY_DOMAINS`	Asks if an EDI variable exists and gets information about it - including domains
`EDI_VAR_LIST`	Gets the list of existing EDI variables in the EDI file
`EDI_VAR_CHECK_SIZES`	Checks if each size of a dependent variable is equal to size of the respective independent variables
`EDI_VAR_EXIST_DOMAIN`	Checks whether an EDI variable has a domain with given name
`EDI_VAR_QUERY_CLASS`	Retrieves the class of an EDI variable
`EDI_VAR_QUERY_ATTR`	Retrieves the *value* of a single attribute
`EDI_VAR_QUERY_ATTRS`	Retrieves the names and values of all attributes
`EDI_VAR_QUERY_COMPONENTS`	Retrieves the number, names and types of one or more components defined in an EDI variable
`EDI_VAR_QUERY_REFS`	Retrieves the references of all the components defined in an EDI variable

Table 6.2

Functions used at OUTPUT	***Purpose***
`EDI_VAR_PUT`	Declares an EDI variable (creating it if needed) – including rank and sizes but excluding domains.
`EDI_VAR_PUT_DOMAINS`	Declares an EDI variable (creating it if needed) - including rank, sizes and domains
`EDI_VAR_PUT_CLASS`	Declares the class of an EDI variable
`EDI_VAR_PUT_ATTR`	Declares a single attribute (name and value) in an EDI variable
`EDI_VAR_PUT_ATTRS`	Declares attributes (names and values) in an EDI variable
`EDI_VAR_PUT_COMPONENTS`	Declares the number, names and types of one or more components in an EDI variable
`EDI_VAR_PUT_REFS`	Declares of one or more reference components in an EDI variable

Table 6.3

Functions used at UPDATE	***Purpose***
`EDI_VAR_REMOVE`	Removes one or more EDI variables from the EDI file
`EDI_VAR_RENAME`	Renames an EDI variable
`EDI_VAR_ADD_DOMAIN`	Inserts one domain and its size in an EDI Variable
`EDI_VAR_REMOVE_DOMAIN`	Removes one domain and its size by an EDI Variable
`EDI_VAR_RENAME_DOMAIN`	Renames a domain defined in an EDI variable
`EDI_VAR_REMOVE_ATTRS`	Removes one or more attributes from an EDI variable
`EDI_VAR_RENAME_ATTR`	Renames an attribute defined in an EDI variable

In the following sections the most important functions will be discussed including experiences obtained so far. The discussion will not present a complete interface - the reader is referred to the EDI user manuals for this information.

6.2.4 Output – creating variables and saving data in EDI files

6.2.4.1 Creating an EDI variable with class and domain specification

Variables are needed when a data set shall be output from to an EDI file. A <Variable> can be created in two ways:

- If the variable is *independent* it can be declared with the function *EDI_VAR_PUT* which simply creates the variable in an EDI file telling its name, rank and <Sizes>.
- If the variable is *dependent*, the function *EDI_VAR_PUT_DOMAINS* can be used. This function has the same arguments plus arguments that specifies the names of the domain i.e. a list of other variables in the file on which the present variable depends. Note, there is no check of consistency i.e. if the domain variables actually exists – it is up to the developer to ensure that all referenced variables exists when the file is closed. A dependent variable can also be created using first EDI_VAR_PUT and then later EDI_VAR_PUT_DOMAINS – this may be adequate in some situations. (Note that a dependent variable may also be created first using EDI_VAR_PUT and the domain specification may then be added later with EDI_VAR_PUT_DOMAINS.)

The call of function EDI_VAR_PUT_DOMAINS is quite characteristic for the EDI interface design:

```
ier = EDI_VAR_PUT_DOMAINS(file_id, name, rank, sizes,
                          dom_n, doms, dom_len)
```

The integer file_id is always there as the first argument followed by the name of the variable (a string – in FORTRAN77/90 declared as CHARACTER(LEN=xx) where xx is the required length). Next rank and sizes information follows and the some data – in this case no. of domains, their names (an array of strings with names of other variables) and the length of those names. Rank and sizes are defined in the usual way e.g. a matrix with 4 rows and 15 columns is rank 2 with <Sizes> 4 15 </Sizes>. Note, that *domains can be specified without the existence of the domain variables* – it is up to the developer to ensure consistency i.e. to add these variables before the EDI file is closed. Also note that the number of domains need not equal the rank (a rather peculiar situation which will rarely appear in practice).

As soon as a variable has been created its class should be declared as discussed in section 5 and according to a Data Dictionary. The class is easily added with

```
ier = EDI_VAR_PUT_CLASS(file_id, name, class)
```

where *class* is a CAHARACTER variable.

Example 5.1: Assume that we want an EDI Variable for a double complex array with a logical flag for each element in the array. The variable shall be named *XXX3D_GridValues* of the imaginary class *FlaggedComplexValuesIn3DGrid* which in an imaginary Data Dictionary depends on the 1D arrays (vectors) X,Y and Z in that order. Assume the integer *file_id* has been assigned a value when the EDI file was opened, that the integer IER has been declared and that the required sizes reside in the integer *NX*, *NY* and *NZ*:

```
INTEGER :: RANK, SIZES(3) ! Obviously rank and sizes
INTEGER :: DOM_N          ! No. of domains
INTEGER :: DOM_LENS(3)    ! Length of domain names
CHARACTER(LEN=20) VAR_NAME
CHARACTER(LEN=30) CLASS_NAME
CHARACTER(LEN=20) DOMAINS(1:3)
   ………
   ! Create EDI variable and set its class
   DOMAINS(1) = ’X’; DOMAINS(2) = ’Y’; DOMAINS(3) = ’Z’
   RANK = 3; DOM_N = 3; DOM_LENS = LEN_TRIM(DOMAINS)
   SIZES(1) = NX ; SIZES(2) = NY ; SIZES(3) = NZ
   VAR_NAME = ‘Our3D_GridValues’
   IER = EDI_VAR_PUT_DOMAINS(FILE_ID, TRIM(VAR_NAME), RANK, &
                    SIZES,DOM_N, DOMS, DOM_LEN)
   IF (IER/=0) GOTO 999 ! Some error reaction at 999
   CLASS_NAME = ‘FlaggedComplexValuesIn3DGrid’
   IER = EDI_VAR_PUT_CLASS(FILE_ID, TRIM(VAR_NAME), &
                           TRIM(CLASS_NAME))
   IF (IER/=0) GOTO 999 ! Some error reaction at 999
   ………
```

Example 5.1 is continued below.

Important remark for FORTRAN90 developers
When a variable name is input to an EDI functions – no matter whether it is the ‘name’ or if it is domain names - there are still some problems with some of the FORTRAN90 functions in how the underlying C++ function gets the data. Not all of these problems has apparently been resolved and an error may be issued from the EDI function or a runtime error may occur. In a given situation the problem can be solved in one of the following ways:

1) In very rare cases one may call the function with the variable name as a character constant e.g. ‘My_FarField’ – it always works.
2) In the majority of cases, however, the argument will be a variable, and in this case *always trim the variable in the call*. If the variable is called VAR_NAME then call the function with TRIM(VAR_NAME) - this usually works.
3) In some cases, however, one may need the length without trailing blanks i.e. set L_V = LEN_TRIM(VAR_NAME) – then call the function with VAR_NAME(1:L_V).
4) In very rare cases none of these option works correctly, and in such cases one can try to copy the trimmed variable name into another character variables and add the null character as last significant character and then call the function with the other variable as argument: V_NAM = LEN_TRIM(VAR_NAME)//CHAR(0) (the CHAR(0) is the string terminator in C/C++ also known as ‘\n’). This tiresome work around usually succeeds when options 2) or 3) fails. One should, however, try 2) first because the code get rather long if 4) is used every time an EDI function is called.

As and example of a combination of 3) and 4) consider the following block of FORTRAN90 statements (we get back to the function EDI_VAR_SET_DOUBLES later – the important issue here is the VAR_NAME argument):

```
IER = EDI_VAR_SET_DOUBLES(FILE_ID, TRIM(VAR_NAME), I, RANK, &
                          V_START, V_COUNT, DOUBLES)
IF (IER<0) THEN
   L1 = LEN_TRIM(VAR_NAME)+1
   V_NAM = TRIM(VAR_NAME)//CHAR(0)
   IER = EDI_VAR_SET_DOUBLES(FILE_ID, V_NAM(1:L1), I, RANK, &
                             V_START, V_COUNT, DOUBLES)
ENDIF
```

Here first an attempt with trim is made and if it fails, the function is called again with the actual argument V_VAM which equals VAR_NAME without trailing blanks concatenated with a CHAR(0) as the last character.

6.2.4.2 Adding features to a <Variable> for data

When a variable for has been created a number of functions can be used to shape it for the specific use. As explained in section 5 there are two main types of <Variable>s: 1) The "normal" type for actual data values and 2) the container that binds a data set together using references.

Besides rank, sizes and domains which are discussed above, the features for the normal <Variable> are the *attributes* and the *components*.

The most important declaration is the components. A <Variable> may have any number of components of one of the available EDI types as shown in table 6.4. The components are declared with function EDI_VAR_PUT_COMPONENTS. All components have the same shape i.e. their rank and sizes are those of the variable (the introduction of structures will remove third limitation. Such developer-defined data types are not available and are not planned to be implemented as static data-typing – the only needed for data exchange – can achieve the same goal by simply nesting component definitions. Each component can be named. This feature is used if a variable has more components.

Table 6.4

Fortran Type	*EDI symbol*	*Description*
	EDI_TYPE_VOID	0 bit
INTEGER	EDI_TYPE_INT	32 bit integer (sign is handled by the application, not by EDI)
CHARACTER*1	EDI_TYPE_CHAR	8 bit character
LOGICAL	EDI_TYPE_BOOL	1 bit logical (8 bit of memory)
CHARACTER*(*)	EDI_TYPE_STRING	arbitrary-length sequence of 8 bit characters
REAL	EDI_TYPE_REAL	32 bit IEEE floating-point real
REAL*8	EDI_TYPE_DOUBLE	64 bit IEEE floating-point real
COMPLEX	EDI_TYPE_COMPLEX	64 bit complex (pair of REALs)
COMPLEX*16	EDI_TYPE_DCOMPLEX	128 bit complex (pair of DOUBLEs)

The function used to specify the components is:

```
ier = EDI_VAR_PUT_COMPONENTS(file_id, name, count, &
                             comp_names, comp_len, edi_types)
```

Where *count* is the number of components, *comp_names* is a CHARACTER array with *count* elements holding the names of the components and *comp_len* an array with the length of the names. Finally, *edi_types* is an integer array (with count elements) that specifies their type.

All attribute values are treated as character strings. Hence a numeric value must be converted from characters to numbers by the tool. The attributes are easy to add using the functions EDI_VAR_PUT_ATTR (one) and EDI_VAR_PUT_ATTRS (several at the same time):

```
ier = EDI_VAR_PUT_ATTRS(file_id, name, count, attr_names,
                        attr_nlens, attr_vals, attr_vlens)
```

Count tells the number of attributes in the character array *attr_names* and then length of these names must recide in *attr_nlens*. Similarly, the corresponding values must be in *attr_vals* and the length of these must be in attr_vlens.

Example 5.1 continued: In the example two components are need – one for the complex values which we call *'OurCmplxCmp'* and one for the logicals called *'Flag'*. Further, the attributes *OurAttib1* and *OurAttrib2* must be added with the values '*Rotated*' and '*Blue*':

```
INTEGER :: COUNT
INTEGER :: COMP_LENS(1:2)     ! Length of attribute names
INTEGER :: COMPTYPES(1:2)     ! For component type specs.
INTEGER :: ATTR_LENS(1:2)     ! Length of attribute names
INTEGER :: ATTV_LENS(1:2)     ! Length of attribute values
CHARACTER(LEN=10) COMPONENTS(1:2)
CHARACTER(LEN=20) ATTIRIBUTES(1:2)
CHARACTER(LEN=20) ATTRIB_VALS(1:2)
   ………

   ! Specify components
   COUNT = 2;
   COMPONENTS(1) = ‘OurCmplxCmp’; COMPONENTS(2) = ’Flag’
   COMP_LENS = LEN_TRIM(COMPONENTS)
   COMPTYPES(1) = EDI_TYPE_DCOMPLEX
   COMPTYPES(2) = EDI_TYPE_BOOL
   IER = EDI_VAR_PUT_COMPONENTS(FILE_ID, TRIM(VAR_NAME), COUNT, &
                             COMPONENTS, COMP_LENS, COMPTYPES)
   IF (IER/=0) GOTO 999 ! Some error reaction at 999

   ! Specify attributes
   ATTIRIBUTES(1) = ‘OurAttrib1’; ATTIRIBUTES(2) = ’OurAttrib2’
   ATTRIB_VALS(1) = ‘Rotated’; ATTRIB_VALS(2) = ‘Blue’
   ATTR_LENS = LEN_TRIM(ATTIRIBUTES)
   ATTV_LENS = LEN_TRIM(ATTRIB_VALS)
   IER = EDI_VAR_PUT_ ATTRS(FILE _ID, TRIM(VAR_NAME), COUNT, &
                         ATTIRIBUTES, ATTR_LENS,          &
                         ATTRIB_VALS, ATTV_LENS)
   IF (IER/=0) GOTO 999 ! Some error reaction at 999
   ………
```

Example 5.1 is continued below.

Assuming that NX=100, NY=100 and NZ=5, the statements in example 5.1 yield the following EDI <Variable> declaration in the file:

```
<Variable Name="Our3D_GridValues"
          Class="FlaggedComplexValuesIn3DGrid" ID="8">
  <Attribute Name="OurAttrib1">Rotated</Attribute>
  <Attribute Name="OurAttrib2">Blue</Attribute>
  <Sizes>5 100 100</Sizes>
  <Domain Reference="Z"/>
  <Domain Reference="Y"/>
  <Domain Reference="X"/>
  <Component Name= "OurCmplxCmp" Type="dcomplex"/>
  <Component Name= "Flag" Type="boolean"/>
</Variable>
```

6.2.4.3 Making a container

A container variable has zero sizes and no domains. Further, it has component references pointing at other variables. It is made quite simple using the function

```
      ier = EDI_VAR_PUT_REFS(file_id, name, count, &
                                 ref_names, ref_len)
```

where count is the number of component references, ref_names is a CHARACTER array with the names of the referenced variables and ref_lens the length of these names.

Example 5.1 continued:

```
   INTEGER :: RANK, SIZES(3) ! Obviously rank and sizes
   INTEGER :: N_REFS         ! No. of ref. components
   INTEGER :: REF_LENS(3)    ! Length of domain names
   CHARACTER(LEN=20) VAR_NAME
   CHARACTER(LEN=30) CLASS_NAME
   CHARACTER(LEN=20) REFERENCES(1:3)
      .........
      ! Create a container
      RANK = 0; SIZES = 0 ! SIZES not used but must be present
      VAR_NAME = ‘Our_Container’
      IER = EDI_VAR_PUT(FILE_ID, TRIM(VAR_NAME), RANK, SIZES)
      IF (IER/=0) GOTO 999 ! Some error reaction at 999
      CLASS_NAME = ‘OurContainerClass1’
      IER = EDI_VAR_PUT_CLASS(FILE_ID, TRIM(VAR_NAME), &
                                TRIM(CLASS_NAME))
      IF (IER/=0) GOTO 999 ! Some error reaction at 999
      REFERENCES(1) = ’X’; REFERENCES(2) = ’Y’; REFERENCES(3) = ’Z’
      N_REFS = 3; REF_LENS = LEN_TRIM(REFERENCES)
      IER = EDI_VAR_PUT_REFS(FILE_ID, TRIM(VAR_NAME), N_REFS, &
                               REFERENCES, REF_LEN)
      IF (IER/=0) GOTO 999 ! Some error reaction at 999
      .........
      ! Set its attributes
      .........
```

As indicated above, a container can of course also have attributes. Finally, an EDI variable may have both its own components based on domains and component references. This quite powerful combination has not been used by the authors but it clearly enables the representation some rather complex data sets.

6.2.4.4 Making a simple variable

In some cases simple variables are needed e.g. as a logical (Boolean) flag. In EDI simple variables are created with EDI_VAR_PUT where *rank equals zero*. Then the class must be declared with EDI_VAR_PUT_CLASS and a single unnamed component is declared with EDI_VAR_PUT_COMPONENTS – the type can be any of the already mentioned EDI-types. The variable value can now be set and saved as discussed in section 6.2.4.5.

6.2.4.5 Transparent IO

Before the functions for saving a <Variable> is introduced, a short discussion about the transparency of input and output is helpful.

Output: During the calculations in the software tool that utilises the EDI, the tool will of course use a number of data structures e.g. arrays for saving intermediate and final results. When data are ready to be saved the relevant EDI PUT function is called with an array with the results as actual input argument. The EDI function will copy the data to the library's own internal data structure and the tools array is free and can be used again or de-allocated. Basically one can consider the PUT operation as a traditional files write statement i.e. when performed, the toll can consider the data as saved.
The actual save to file operation is managed by the EDI Library and the developer need not consider any details. The time and method of output implemented in the EDI may even be changed in future versions. The only thing to remember is that the EDI_FILE_CLOSE function must be called before another program can access the file (e.g. a post processor).

Input: The EDI load data from the file into its own data structures at a time which can't be controlled by the software tool, and again the time and method is of no importance to the tool. When the relevant GET function is called, the EDI will ensure that the required data can be retrieved from the file and become available in the tools own arrays.
Further, since declarations functions (see below) can return all information about a <Variable> including it's <Sizes> one may preferably use *dynamic allocation* by first asking for <Sizes>, allocate and then GET the data. In this way the arrays will always have the correct sizes for the data from the file.

6.2.4.6 Saving values in an EDI <Variable>.

When a variable that has been declared as explained above the tool can SET the variable, i.e. *the tool can save data*. First, the data will be stored in EDI's private data structure. The data will then be saved in the file when EDI_FILE_CLOSE is called. There are two sets of SET functions – one for scalar variables and one for arrays. The functions are:

Table 6.5

SET scalar EDI variable	***SET array EDI variable***
EDI_VAR_SET_INT	EDI_VAR_SET_INTS
EDI_VAR_SET_BOOL	EDI_VAR_SET_BOOLS
EDI_VAR_SET_CHAR	EDI_VAR_SET_CHARS
EDI_VAR_SET_REAL	EDI_VAR_SET_REALS
EDI_VAR_SET_DOUBLE	EDI_VAR_SET_DOUBLES
EDI_VAR_SET_COMPLEX	EDI_VAR_SET_COMPLEXES
EDI_VAR_SET_DCOMPLEX	EDI_VAR_SET_DCOMPLEXES
EDI_VAR_SET_STRING	EDI_VAR_SET_STRINGS

They all look alike - except for the variables for EDI_TYPE_STRING (the reader is referred to [14] for details on EDI_STRINGS). Here we show the interface to the EDI_VAR_SET_INTS for setting an EDI integer array variable. This function sets or changes all or a part of the INTEGER values of an EDI variable.

```
        ier = EDI_VAR_SET_INTS(file_id, name, comp, rank,
                               vstart, vcount, vals)
```

where

`ier:`	Returned value. It is equal to an integer >= 0 if the operation is successful, otherwise < 0.
`file_id (IN):`	Integer that contains the EDI file identifier.
`name (IN):`	String that contains the variable's name.
`comp (IN):`	Integer that contains the index of the component to use.
`rank (IN):`	Integer that must contain the variable's rank.
`vstart (IN):`	Integers array, with length equal to `rank`, that contains the start indices (one for each dimension).
`vcount (IN):`	Integers array, with length equal to `rank`, that contains the number of values to set or change (for each dimension).
`vals (IN):`	INTEGERS array that contains the values to set.

As discussed above, a variable can have several components. The `comp` argument points out which component is set in the present call and its type must of course be consistent with the SET function which is called i.e. in this case it must be an integer component. Next, note that the function is designed such that one can set the whole array or a sub-array using `vstart` and `vcount.`

The most "sensible" input is the `vals` argument. *Note, that there is no specific description of the declared shape and size of* `vals`*.* Hence `vals` can be an integer array declared to have any dimension what so ever and with any sizes. All that is transferred to the SET function *is the start address of the array*. All the values to be transferred are then assumed to be in the following **vcount(1)** × **vcount(2)** × **...** × **vcount(rank)** places in the memory.

Assume that **rank**, **vstart**, **vcount** and **vals** has been declared as follows:

```
   INTEGER :: RANK, VSTART(1:2), VCOUNT(1:2)
   INTEGER :: VALS(1:7,1:5)
```

```
……
RANK = 2
VSTART(1) = 1; VSTART(2) = 1
VCOUNT(1) = 7; VCOUNT(2) = 5
……
! Assignment values to integer array vals
```

In this case there will be no problem – all sizes fits and the values that has been put into vals will be saved as intended. If, however, only the first 5 rows are used (e.g. the array is reused for a smaller integer matrix later) we have a problem, since FORTRAN90 stores data column wise as shown in figure 6.1 -

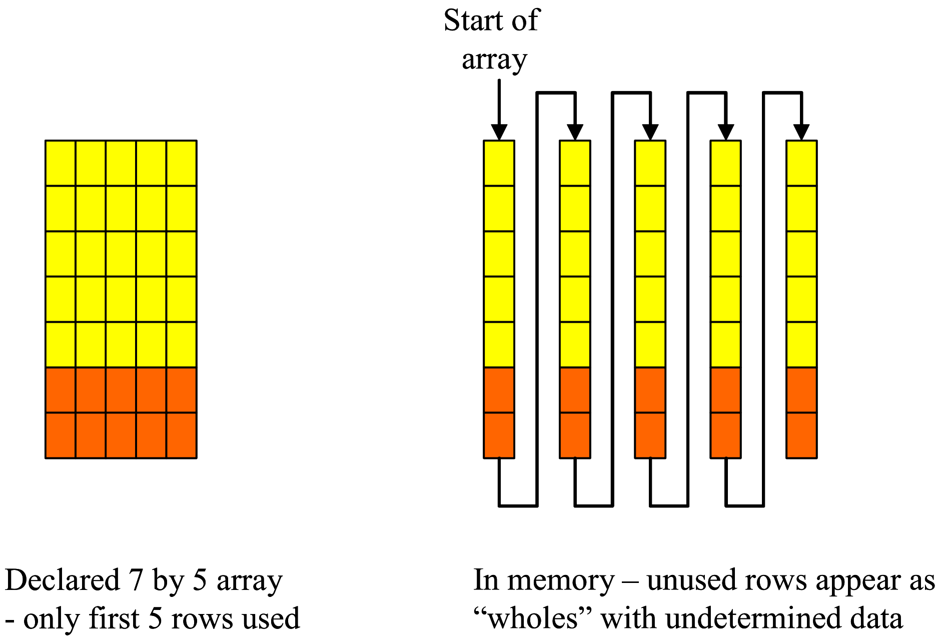

Start of
array
Declared 7 by 5 array
- only first 5 rows used
In memory – unused rows appear as
“wholes” with undetermined data

Figure 6.1: FORTRAN90 actual storage of 2D static array in memory

The same situation appear for static arrays in C/C++ but now with columns since data are stored row wise in memory. The easiest way to avoid the above problems in to use dynamic arrays and allocate exactly what is needed before a SET function is called (C/C++ developers do this but many FORTRAN90 developers tend to use static arrays).

Further details about the various functions in section 6.2.4 can be found in [14].

6.2.5 Input of data from EDI files

6.2.5.1 Obtaining information about EDI variables

When an EDI file has been opened, a software tool needs to know if the content of the file is as expected (i.e. if a far field is read) the tool needs to know if all the required variables are present. There may be many other data in the file but this is of no importance here – the tool will only load the data it needs.

The first thing always is to get a list of the EDI variables in the file – this is obtained with the important function

```
   ier = EDI_VAR_LIST(file_id, count, names, name_len, reqlen)
```

The count argument tells the function the length of the *names* string array at call. Quite obviously the function returns the number of variables in the file in *count*, and the names of the variables. The names array should be declared dynamically e.g. to 20 names. If this is too short then ier>0 and count>20 at return and it is easy to re-allocate the array to the correct size and call the function again.

When the names have been found one can start to examine all the other features of the file. For this purpose there is a number of EDI functions as listed in table 6.6. First, the tool will need to locate the data set root containers.

Table 6.6

Functions used at INPUT	***Purpose***
`EDI_VAR_QUERY`	Asks if an EDI variable exists and gets information about it - *not* including domains
`EDI_VAR_QUERY_DOMAINS`	Asks if an EDI variable exists and gets information about it - including domains
`EDI_VAR_LIST`	Gets the list of existing EDI variables in the EDI file
`EDI_VAR_CHECK_SIZES`	Checks if each size of a dependent variable is equal to size of the respective independent variables
`EDI_VAR_EXIST_DOMAIN`	Checks whether an EDI variable has a domain with given name
`EDI_VAR_QUERY_CLASS`	Retrieves the class of an EDI variable
`EDI_VAR_QUERY_ATTR`	Retrieves the *value* of a single attribute
`EDI_VAR_QUERY_ATTRS`	Retrieves the names and values of all attributes
`EDI_VAR_QUERY_COMPONENTS`	Retrieves the number, names and types of one or more components defined in an EDI variable
`EDI_VAR_QUERY_REFS`	Retrieves the references of all the components defined in an EDI variable

First, the tool will need to locate all root containers (see section 5.2.5) of the class of the variable that represents the data set to be used. Assume all names of variables have been put in a list and that we are looking for a far field i.e. we must locate all variables of class *Field:Far*. Further, assume that we will use the first far field located unless a specific name is given in which case this field shall be used. This is easy:

```
   Loop through the variables in the list
      Fetch the class of the present variable
      If class equals ‘Field:Far’ then
         If we are looking for a specific named variable then
            FOUND= Name of present variable equals name searched for
         Else
            FOUND = .TRUE.
            Remember name of variable
         If FOUND exit variables loop
```

- or in FORTRAN90 – as a general function that may look for variables of a specific class:

```
FUNCTION LOCATE_VAR_OF_CLASS(EDI_FILE, COUNT, NAMES,
                             CLASS, VAR_NAME)

   ! IF VAR_NAME non-blank, look for variable with that name
   ! of class CLASS in EDI_FILE.
   ! IF VAR_NAME is blank at call, return first EDI variable
   ! of class CLASS found in EDI file and its name in VAR_NAME.
   ! The file must be opened. All names of EDI variables in
   ! the file must be in the names array NAMES at call.
   ! Function value is 0 if a variable is found, Otherwise false.

   INTEGER, INTENT(IN) :: EDI_FILE
   INTEGER, INTENT(IN) :: COUNT
   CHARACTER(LEN=MAX_NAME_LENGHT), INTENT(IN) :: NAMES(1: COUNT)
   CHARACTER(LEN=MAX_NAME_LENGHT), INTENT(IN) :: CLASS
   CHARACTER(LEN=MAX_NAME_LENGHT), INTENT(INOUT) :: VAR_NAME
   INTEGER :: LOCATE_VAR_OF_CLASS ! Function value

   ! Locals
   INTEGER :: I, IER, CLASS_LEN
   LOGICAL :: FOUND
   CHARACTER(LEN=MAX_NAME_LENGHT) :: VAR_CLASS

      LOCATE_VAR_OF_CLASS = 0 ! Assume success
      DO I=1, COUNT
         IER = EDI_VAR_QUERY_CLASS(EDI_FILE, TRIM(NAMES(I)), &
                                   VAR_CLASS, CLASS_LEN)
         IF (IER/=0) GOTO 10 ! Exit if problems (my be refined!)
         IF (VAR_CLASS == CLASS) THEN
            IF (VAR_NAME/=’ ‘) THEN
               FOUND = VAR_NAME == NAMES(I)
            ELSE
               FOUND = .TRUE.
               VAR_NAME = NAMES(I)
            ENDIF
            IF (FOUND) GOTO 999 ! Exit the DO loop
         ENDIF
      ENDDO
      ! We end up here if no variable of CLASS.

      10 CONTINUE
      LOCATE_VAR_OF_CLASS = -1

   999 RETURN
END FUNCTION LOCATE_VAR_OF_CLASS
```

When the entry container has been located all sorts of checks and cross-checks can easily be made using the functions listed in table 6.6 and the consistency of the data set can be verified. Cross-checks include

- Does all directly of indirectly referenced variables exist (this check can be made with a recursive function starting with the entry container and working down through references).
- Are the sizes of an EDI variable with domains consistent with the sizes of its domain variables. The EDI function `EDI_VAR_CHECK_SIZES` can be used for this check.
- Are all mandatory attributes of an EDI variable present and are their values as expected.

The general rule should be that the input module which fetches data from the EDI file checks the validity and consistency of a variables features before the actual data are fetched.

6.2.5.2 Getting the values of an EDI variable

Following checks, the tool reading the EDI file will need to fetch the values (i.e. the content of the <Data>-section) for each of the required variables. Here the GET functions are used:

Table 6.6

SET scalar EDI variable	***SET array EDI variable***
EDI_VAR_GET_INT	EDI_VAR_GET_INTS
EDI_VAR_GET_BOOL	EDI_VAR_GET_BOOLS
EDI_VAR_GET_CHAR	EDI_VAR_GET_CHARS
EDI_VAR_GET_REAL	EDI_VAR_GET_REALS
EDI_VAR_GET_DOUBLE	EDI_VAR_GET_DOUBLES
EDI_VAR_GET_COMPLEX	EDI_VAR_GET_COMPLEXES
EDI_VAR_GET_DCOMPLEX	EDI_VAR_GET_DCOMPLEXES
EDI_VAR_GET_STRING	EDI_VAR_GET_STRINGS

Once again they are all alike (with the exception of the string handler) – the only thing that changes is the type of the VALS array (last argument), which of course changes with the specific type of the data to be fetched from the EDI-file:

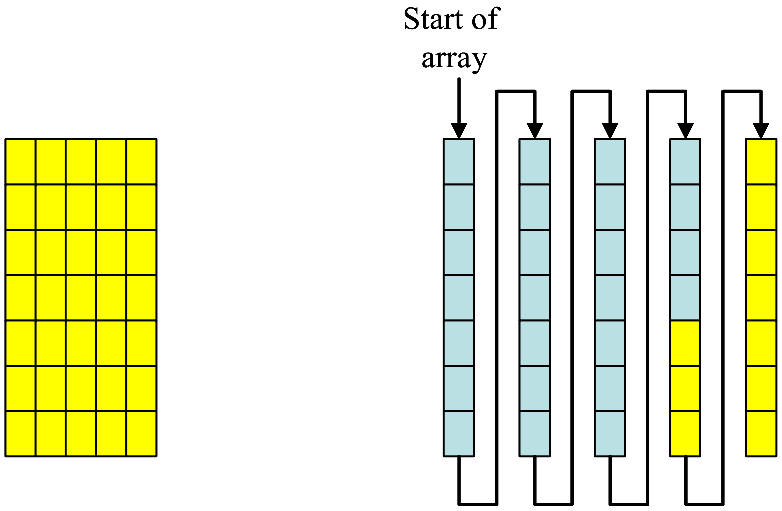
```
ier = EDI_VAR_GET_INTS(file_id, name, comp, rank,
                       vstart, vcount, vals)
```

where

`ier:`	Returned value. It is equal to an integer >= 0 if the values are successfully returned, otherwise < 0.
`file_id (IN):`	Integer that contains the EDI file identifier.
`name (IN):`	String that contains the variable's name.
`comp (IN):`	Integer that contains the index of the component to use.
`rank (IN):`	Integer that must contain the actual number of variable's dimensions (rank).
`vstart (IN):`	Integers array, with length equal to `rank`, that contains the start indices (one for each dimension).
`vcount (IN):`	Integers array, with length equal to `rank`, that contains the number of values to retrieve (for each dimension).
`vals (OUT):`	INTEGERS array that will contain the retrieved values.

The component must of course be of the correct EDI_TYPE i.e. EDI_TYPE_INT in the above case.

Once again the developer should take care with the array dimensions. The functions return **vcount(1)** × **vcount(2)** × **...** × **vcount(rank)** values that reside densely in memory.
Assume the function is called with a static integer array where declared as 7 rows and 5 columns. Further, assume that vcount(1) = 5 and vcount(2) = 5 and we want to fetch these 25 values from a rank 2 EDI variable into the array. Then at return the data values reside in the array as shown in figure 6.2:



Declared 7 by 5 array

Returned values will occupy the first 25 places in the array corresponding to the first 3 columns and 4 top places of 4th column.

Figure 6.2: FORTRAN90 location of contiguous array elements in memory

Many problems will be avoided if dynamic allocation is used such that the array exactly fits with the data to be fetched. In the following example assume that the second component is going to be fetched from a rank 3 EDI variable with <SIZES> 17 201 10 </SIZES>, and assume that this is a complex:

```
! Declaration of
COMPLEX, ALLOCATEBLE :: VALS(:,:,:)
...........
   VSTART = 1 ! Set all equal to 1
   VCOUNT(1) = 17; VCOUNT(2) = 201; VCOUNT(1) = 10;
   RANK = 3; COMP = 2
   VAR_NAME = ‘My_CmplxField’
   IF (ALLOCATED(VALS)) DEALLOCATE(VALS)
   ALLOCATE(VALS(17,201,10))
   IER = EDI_VAR_GET_ COMPLEXES(FILE_ID, TRIM(VAR_NAME), COMP,
                                RANK, VSTART, VCOUNT, VALS)
   ...........
```

Here, the array VALS fits exactly with the data and the application may address the elements accordingly.

7. Final remarks

In this document we have only covered the most fundamental part of the EML language and EDI level 0. Much more can be said about the both and a later edition of the present document will cover some of the key advanced features of EML and key functions of EDI level 1. Until this extension appears the user is referred to the user manuals.

In EML, more options are possible in the declaration of variables, allowing the nesting of components, to declare data structured similar to what can be obtained with derived types (e.g. `struct` in C/C++, `TYPE` in FORTRAN90, `record` in Pascal). However the corresponding EDI functions have been added recently and not used enough by the authors to discuss them.

The EDI is a nice interface that has been carefully planned. Further, much of the underlying implementation has been tested by a number of ACE and EAML partners and those parts of the library perform satisfactory. There are, however, still parts of the library especially in level 1 that have only been tested by ITLink. It is hoped that more users will look into and test these parts in the near future.

Several proposals for future improvements and if implemented they will be covered in future editions of the present document.

8. References

[1] http://www.antennasvce.org/

[2] Sabbadini M. et al, European Antenna Modelling Library: An Open Platform for Space Antenna Engineering

[3] http://en.wikipedia.org/wiki/OSI_Model

[4] http://en.wikipedia.org/wiki/XML

[5] Sabbadini M., Data File Format for EM Modelling, 1994, (XEA/098.94, rev.1).

[6] Rew R. and Pincus R., NetCDF User’s Guide for Fortran90Edi manual, An Access Interface for Self-Describing, Portable Data, Version 3.5, March 2002 (http://www.unidata.ucar.edu/).

[7] Sabbadini M., System Analysis and Requirements for an Electromagnetic Data Exchange Standard, ESTEC Working Paper No. 2330 Issue 1, ESTEC, Noordwijk, The Netherlands, October, 2005.

[8] Frandsen P.E., Ghilardi M., Sabbadini M., Benvenuti L., Freni A., Mioc F., Martin T., Martinaud J.-P., Rylander T., Yang J,: Requirements for the Electromagnetic Data Interface (EDI) - XML file format and FORTRAN API, Joint EAML/ACE ASI File Formats activity, October 2005.

[9] Frandsen P.E., Sabbadini M., Ghilardi M., Silvestri F., Proposal for the Electromagnetic Data Interface (EDI) - XML file format and FORTRAN API, Joint EAML/ACE ASI File Formats activity, October 2005.

[10] Baker, F. (contact), HDF5 User's Guide, HDF5 Release 1.6.5, November 2005 (http://hdf.ncsa.uiuc.edu/HDF5/)

[11] Martinaud J-P., Frandsen P.E., Vandenbosch G., The Ace Activity On Standardized File Formats For Electromagnetic Software, Proc. EuCAP 2006.

[12] Mioc F. (ed.), Field Data Dictionary, ACE- 2, Document Number FP6-IST-508009.

[13] Volski V. (ed.), Currents DD, ACE- 2, Document to be released.

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

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

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

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

[18] P. E. Frandsen, M. Ghilardi, F. Mioc, M. Sabbadini, F. Silvestri. The Electromagnetic Data Exchange: Much More Than A Common Data Format, Proc. EuCAP 2007.