Aristotle University of Thessaloniki
Department of Physics
Section of Astrophysics, Astronomy and Mechanics
Laboratory of Astronomt

Conversion of pulsars data to XML (eXtensible Markup Language)

Diploma Thesis
by
Radioti Ekaterini

Supervised by: J.H. Seiradakis

Thessaloniki, June 2002

PART II

Part II

1.EPN FORMAT

A flexible format for exchanging pulsar data

The **E**uropean **P**ulsar **N**etwork (EPN) is an association of European astrophysical research institutes that co-operate in the subject of pulsar research. All institutes have up until now developed their own individual hardware and software facilities tailored to their own requirements and will, of course continue to do so in the future. Contact and co-operation has always existed between the scientists of the member institutes and outside, but the lack of a common standard format for pulsar data has hampered previous collaborative research efforts.

The following pages describe the structure of the format and demonstrate a simple pulsar analysis done with software that implement reading/writing formats.

The EPN format

The underlying principles of the format result from a number of requirements. This was essentially a balance between the need for efficient data storage and providing sufficient information about the data for potential users. Specifically, the following requirements had to be met:

- **Operating system independence**: The data format should be as portable as possible between present and future operating systems, therefore it uses only ASCII-data throughout. These data have been arranged so that words are aligned over 80-byte boundaries, this simplifies inspection and printing of the files.
- **Completeness**: The data should contain all information for the identification of the source and the observing circumstances useful for further analyses of the data by others.
- **Compactness**: Descriptive information should not dominate the format. The measured values that form the bulk of a block of data are given as scaled four-character hexadecimal numbers, giving a dynamic range of 65536:1.
- **Versatility**: The format should be suitable for continuously sampled multi-channel filterbank search data, synchronous integrated and single-pulse data as well as processed data. Space is left for more descriptors, future adaptations and expansions.
- **Simplicity and ease of access**: The data format consists of a standardized fixed-length header with a variable length data structure attached to it. The header fully describes the structure of the data, which is not changed within one file but can vary between files. In this way it is possible to calculate the length of the data block within each file after reading its header. The file can then be opened for random access with fixed block length, faster than a sequential read.

Each EPN file consists of one or more EPN blocks. The basic structure of an EPN block is shown in the following figure. Each file has a common fixed length *header* followed by a number of individual *data streams* of equal length (figure 1). The header describes the data, containing information on the pulsar itself, the observing system used to make the observation as well as some free-form information about the processing history of the data. The onus is on the site-specific conversion process to ensure correct conversion to the standardized entries and reference to common catalogues.

The data streams themselves may be outputs of different polarization channels, or individual channels (bands) of a filterbank or a combination thereof. Each data stream starts with a small, fixed length sub-header in front of the actual data values. The number of data streams and their length may vary between different EPN files, but is constant within each file. A character field and an ordinal number are provided for each stream for its identification.

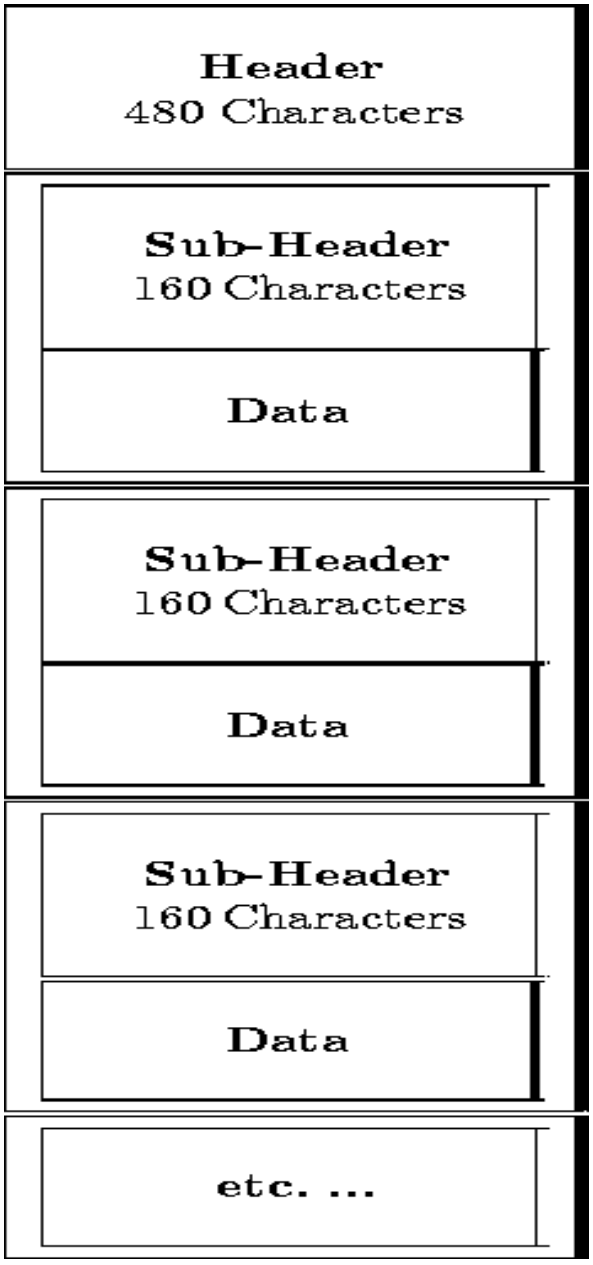

Figure 1: Schematic representation of an EPN data block.

EPN header

The first 80-character line of the header, consists of the following parameters:

First line:

1. *version* (8 ASCII characters). Stores the version of the EPN format used in the file.
2. *counter* (4-digit integer). The number of 80-character lines that constitute the record.
3. *history* (68 ASCII characters). History of the data and various comments, preferably meaningful notes are stored in the parameter.

Second line:

4. *jname* (12 ASCII character). Stores the name of the pulsar derived from its J2000 coordinates.
5. *cname* (12 ASCII characters). Represents the common name of the pulsar.
6. *Pbar* (16-digit float number). The current barycentric period of the pulsar, measured in seconds.
7. *DM* (8-digit float number). Represents the dispersion measure, measured in parsecs per cubic meter.
8. *RM* (10-digit float number). Represents the rotation measure, measured in radians per square meter.
9. *Catref* (6 ASCII characters). Indicates the pulsar parameter catalogue in use.
10. *Bibref* (8 ASCII characters). Indicates the bibliographic reference for data.
11. *–empty space--*. 8 characters of empty space are left between the second and the third line of the header for expansion.

Third line:

12. *rah* (2-digit integer). Indicates the hours of right ascension according to the J2000 astronomical catalogue.
13. *ram* (2-digit integer). Indicates the minutes of right ascension according to the J2000 astronomical catalogue.
14. *ras* (6-digit float). Indicates the seconds of right ascension according to the J2000 astronomical catalogue.
15. *ded* (3-digit integer). Indicates the degrees of declination according to the J2000 astronomical catalogue.
16. *dem* (2-digit integer). Indicates the minutes of declination according to the J2000 astronomical catalogue.

17. *des* (6-digit float). Indicates the seconds of declination according to the J2000 astronomical catalogue.
18. *telname* (8 ASCII characters). Name of the telescope or location of the observatory.
19. *epoch* (10-digit float). The modified Julian Date of observation, measured in days.
20. *opos* (8-digit float). The relative or absolute polarization position angle of the telescope feed system, measured in degrees.
21. *paflag* (1 ASCII character). Defines whether the absolute polarization PA is available or not. If it isn't, there is a possibility for the relative polarization to exist or neither of the above is displayed.
22. *timflag* (1 ASCII character). Defines whether time stamp represents barycentric (B) or topocentric universal time (U).
23. –*empty space*--. 31 characters of empty space are left between the third and the fourth line of the header, for expansion.

Fourth line:

24. *xtel, ytel, ztel* (three 17-digit floats). Topocentric rectangular position of telescope, measured in meters.
25. –*empty space*--. 29 characters of empty space are left between the fourth and the fifth line of the header, for expansion.

Fifth line:

26. *cdy* (4-digit integer). Indicates the year of creation or modification of the dataset.
27. *cdm* (2-digit integer). Indicates the month of creation or modification of the dataset.
28. *cdd* (2-digit integer). Indicates the day of creation or modification of the dataset.
29. *scanno* (4-digit integer). Scan number is the serial number of the observation.
30. *subscan* (4-digit integer). Defines the sub-sequence number of the observation.
31. *npol* (2-digit integer). Defines the number of the polarization channels observed.
32. *nfreq* (4-digit integer). Defines the number of frequency bands per polarization.
33. *nbin* (4-digit integer). Indicates the number of phase bins per frequency.
34. *tbin* (12-digit float). Is the required sampling interval to write the information included in a bin, measured in μsec.
35. *nint* (6-digit integer). Indicates the number of integrated pulses per data block.

36. *ncal* (4-digit integer). Defines the bin number of start of cal signal, measured in µsec.
37. *lcal* (4-digit integer). Defines the number of bins in the cal signal, measured in µsec.
38. *tres* (12-digit float). Defines the temporal resolution.
39. *fluxflag* (1 ASCII character). Defines whether data is calibrated in terms of flux or not. “F” signifies flux-calibrated data.
40. –*empty space*--. 15 characters of empty space are left between the fifth and the sixth line of the header, for expansion.

Sixth line:

41. *line 40, line 40* (80 ASCII dashes). The sixth line contains no information, is used to separate the header from the sub-header.

EPN sub-header

The sub-header lines are attached to every data stream and contain parameters concerning the respective stream.

First line:

1. *idfiend* (8 ASCII characters). Describes the data stream using the stokes parameters I,Q,U,V.
2. *nband* (4-digit integer). Ordinal number of the data stream.
3. *navg* (4-digit integer). Defines the number of streams averaged into current one.
4. f_0 (12-digit float). Effective center sky frequency of this stream.
5. f_{0u} (8 ASCII characters). String giving unit of f_0.
6. *df* (12-digit float). Effective bandwidth of this stream.
7. *dfu* (8ASCII characters). String giving unit of df.
8. *tstart* (17-digit float). Indicates the time elapsed since the beginning of the current Julian day, measured in µsec.
9. –*empty space*--. 7 characters of empty space are left between the first and the second line of the sub-header, for expansion.

Second line:

10. *scale* (12-digit float). Scale factor for the data.
11. *offset* (12-digit float). Offset to be added to the data.
12. *rms* (12-digit float). The rms value for the data stream.

13. *papp* (16 digit float). Indicates the apparent period at the time when the first beam is recorded, measured in sec.
14. *–empty space*--. 28 characters of empty space between the second line of the sub-header and the EPN data, for expansion.

EPN data

The data part of an EPN record is written in hexadecimal format. The number of characters per record is (6+maxblk*(manbin/20+2)*80, where maxblk is the maximum number of data streams including in a record and maxbin is the number of bins, of which each data stream consists.

Below is shown a part of an EPN record of the Pulsar PSR 1929+10.

```
EPN 6.00 222 Data created by epos2epn v6.0
1932+1059 1929+10 .2265171530 3.176 -6.100tmlc95 studen19
193213.900+105931.994effberg 49575.000 .000
4033949.50000 486989.40000 4900430.80000
2000511212110000 4 11024 221.000000 50 .000000 0 1 0
------------------------------------------------------------------------------
LHC 1 1 .000000 GHz 40.000000 MHz 14578478479
.100000E+01-.441750E+05.300701E+02 .226518094540
31B595F3F1A2E8CCFA32E8CCE9A0E9A0F115EEDFFD82EDC5F088EF26F728F3D8FFFFEDC5ECAAECF0
ECF0E7F8F276DB8BDCECEB02EFFADD33F4F3E2FFF4ACF6E2ECAAF1A2F465ED37F4F3EFB3EEDFF5C7
E9A0F465EFB3ECAAF580EE99F276F654E4A8EFFADD7A6A981F7B1B5622CB12390E1509F119AE0C26
08D6109116A513541E600E1501611617139B146F26A914B62C7628DE1A3C0BDF26A90F301C7128DE
0B99123929F91A3C223E23120F301B9D1BE417BF19F51C7112C7116514B610D817320B5223121968
1CFF223E1C7120961732058506A017BF2F7F0EE9042423120E5C14FC161718931C710CFA1E601921
130E072E1B10242D14B6184D20960A7E1B101B56104B17BF26A9072E091C09F11A82184D0F761E60
184D1C2A1FC20D411EED184D16EB235912391BE41F7B1F342008154314B616A519AE08D6109121F7
12C70A3721B020DC173210041617088F212312801F7B180623E617321AC90EE9280A239F12C71921
154311F31FC215430EE92E1E26A91BE4142822CB1D45142809F11D4513E20A7E0D41184D0D8817BF
204F239F165E0BDF0963165E18060AC51CFF111F0FBD2A4010041E19111F1617116521231F3415D0
142826F019AE21B015D01F34139B130E1BE40FBD14281CFF0E5C1893184D1AC918931EED216A216A
2359216A0EE90B991DD3154318DA1FC214FC21B01D8C14281EA71CFF15D01F341AC90DCE0D41300D
21230D8812390C261BE41A8217BF184D22842736139B2898130E1004104B1C2A1F7B0E152B5B1806
158A19F51E191B100FBD000013E21B56135412391D4511651EED011A1C2A1091111F14280DCE184D
1B9D18DA10912C76116517BF1BE41E6029B320DC146F1B1020961BE417791CFF2123209610911806
0E5C19F51BE41C711A821BE403971A821B101B9D20DC1F3423E61C710F761AC9192113E211F31354
1C2A1B101B9D10910EE9239F184D1B56111F1C2A19AE15430E5C1E191F3419AE16A514B60C6D2096
242D1B561E601E19189319F50EE908D6088F16A5204F146F196810D81C2A15431E19204F11651239
111F0EA21732189300D415D01B9D18DA25D506A0242D1C71177912C719F51A3C1732123916171CB8
139B146F21F71CFF165E1E1912C719F514FC111F20960B5214FC1D8C12C71E1915431F7B2CBC0BDF
1F7B1B100E5C1FC21D8C250118931428280A223E184D100417791EED084806A01F7B165E1CB82123
111F1D4511F3091C139B1B9D00461B9D1D8C2473184D0EE91B5617BF0E5C1B5613541D451DD318DA
1893184D109106A009AA1B9D146F223E19AE0EE91C7111650EA229B30D8826A917BF14FC1EED1239
19210CB4189309AA1D4510041091173221F71EA70CFA21230E15111F111F29B30FBD2ACD17791B10
0F3007BB20DC12391FC21AC917BF1B56146F0AC521B029B31893239F20DC142819212925146F2B14
2D03189316EB196816171E600FBD196818930B0B1C2A12C715D0158A1806142808480DCE0D8835D9
13E214FC1AC91B5620DC09F117790EA20A371CB825D519F51A3C1B102736258E142813E220081893
1A8223E6072E17BF14FC1EED1280239F1C2A24BA09F11CB8189314FC24BA10041A3C2DD71D451AC9
19F5165E1165223E29251EA728511D451893139B16A51CB8223E1AC90C6D273619681E600EA207BB
1D451D8C242D29251A3C1F7B19F5046B16EB228426F02312111F1091184D2096239F1BE42A870EE9
223E21B01F7B189320DC2096261C14B60D412D0311F321F70F3026F02CBC1428158A250124BA2925
17321B1012C71B561A3C1B9D19F51EED25012C2F296C266226A93D95385545973CC1405847CC34BE
58A563235ADA59C052D851BD57D15D575BAF617B61C2697D73B5720D851B66BA6EBD50A35B684F88
3F3D5C3C44C347CC39FE316E343133EA35D9362025D52F38389C2501335D2E1E2096378113541617
242D26A914B630E120081CB8173221B0231218931EED2CBC1AC91E60130E18DA06E706E70DCE21B0
1BE41AC90D88123911AC184D1C2A29251F7B1D4523E609AA09630B5215431FC21FC212391D451F7B
10D811AC1C711E191B56135423121617158A22840CB4072E109109AA16A512C71968204F15430963
14280E150E5C1D8C10D80EE9192120DC0CFA139B1732223E11AC31B5192122841A3C109117BF0B52
1AC91B9D23E618DA173218DA228410D82BA1139B084814B6139B0E151543277D14B61D8C1CFF2736
16171B9D1B5624730D880C261C711EED14B617BF2C2F0DCE22CB21F71B5611F31EA71E60139B1A82
1617135415D016A50C6D123909F124731A3C200813541CFF13E2239F1239109128981AC920DC14FC
130E1BE417BF13E21F7B2ACD1B5620962359158A158A239F1806139B18060EE9161720081004146F
11F319211F3423591F7B16EB13E229251EA71F3419AE0D88204F192112391E19111F19AE0EA21921
1D8C0EE919210CFA24BA0E5C130E1C7113E220DC1A3C1C2A200820DC14FC1732000016EB03DD15D0
1B560FBD1CFF216A07741280277D15D0161714B623591D8C1EA71A3C23121DD30DCE1A3C0DCE1E19
2D031F7B19211D451F3421F7165E11651921216A16A504F81DD310D8189319681968196819F51B9D0CFA
184D200814FC21F711F314FC11F30E150FBD0B0B12801B9D07740EA2158A0B52165E0B52184D1E19
139B0EA21C2A173216A52B140D4118DA1617123910D8111F139B0E5C1CB8146F0A37096311AC1C71
```

```
15430F301B9D1FC20EE91004200805CC104B18DA19AE0C261E1920DC0E5C18DA11F3177916A5091C
06A0216A2925123914B615431FC2231222840EA2261C19AE165E209613E212801C2A1D8C1AC9158A
1B9D0D88128001A80000000000000000000000000000000000000000000000000000000000000000
```

2.XML (eXtensible Markup Language)

The demands on electronic data processing on the Web as well as in other publishing and processing media can be reduced to a common denominator: the access to information as well as information processing has to be *simple*, *targeted*, *flexible* and *independent*:
Simple, that is largely automated and thus faster, less expensive and of consistently high quality *targeted* and *flexible*, according to the information and its purpose, *independent* from the medium, i.e., compatible with any kind of hard- and software.

In order to fulfil these requirements, SGML, a well-established standard, was developed. SGML is the origin of XML, which was created to enable information-oriented data processing on the Web. In this basic demand both languages are similar.

The experience with the standard SGML was used for the development of XML and is of great value for its use.

Current Approach: SGML, a Silent Revolution

SGML (Standard Generalized Markup Language), an international standard since 1986, is a meta-language: it can be used to create new languages in order to describe any kind of information.

A document contains two kinds of information: content and structure. A bibliographical entry may have the following content:

> Charles F. Goldfarb/ Steve Pepper/ Chet Ensign: SGML's Buyer's Guide
>
> A unique guide to determining your requirements and choosing the right SGML and XML products and services
>
> ISBN 0-13-681511-1

This content consists of structural units (elements) "author, title, subtitle, ISBN" the entries of which come after each other. Each author consists of the last name followed by the first name.

Using SGML content and structure can be described in a standardized way:

```
<biblio-entry id="12">

<author>
<last-name>Goldfarb</last-name>
<first-name>Charles F.</first-name>
</author>

<author>
<last-name>Pepper</last-name>
<first-name>Steve</first-name>
</author>

<author>
<last-name>Ensign</last-name>
<first-name>Chet</first-name>
</author>

<title>SGML's Buyer's Guide</title>

<subtitle>A unique guide to determining your requirements and choosing the right SGML and XML products and services</subtitle>
```

<isbn>0-13-681511-1</isbn>
</biblio-entry>

The general definition of this structure is determined in the Document Type Definition (DTD):

<!DOCTYPE biblio-entry
<!ELEMENT biblio-entry - - (author+, title, subtitle?, isbn)>
<!ATTLIST biblio-entry id CDATA#REQUIRED>
<!ELEMENT (last-name, first-name, subtitle, isbn) - - (#PCDATA)
>

It represents the structure rules, the building plan for all documents or document parts of the same type (e.g. bibliography, report, memo). A document is only valid SGML if it complies with the structure rules of DTD.

The advantages compared with layout-oriented format are huge. They include:

Access to information: since the documents are structured with regards to content (in the example author, title, ISBN etc.), the information parts and its relationship are available electronically and usable for various purposes and in different ways, e.g., for retrieval. In the example it is possible to formulate a search request which does not search "Pepper" via full text retrieval but as the last name of an author within a bibliographical entry. This is different than "Pepper" as a dictionary entry, which could be structured as follows:

<diction-entry>

<lemma id="1076">Pepper</lemma>

<definition no="1">Pepper is a hot-tasting spice which is used to flavor food</definition>

<definition no="2">A Pepper is a hollow green, red, or yellow vegetable with seeds. See pictures headed
<link ref="11983">vegetables</link>
</definition>

</diction-entry>

In the full text retrieval, both occurrences of "Pepper" are counted as hits; a structured search enables a distinction of different information. (What could be more different than authors and spices!).

Single source, multiple outputs: A single data source can be used in different ways for different media. Since the source has a clear structure, the conversion can be automated.

Links, for example, can be emphasized in the print only in the format (e.g. italic) whereas they could be hyperlinks on a CD-ROM. Thus, independence from hard- and software is ensured: if for example, the software used for printing changes, the conversion of this output format is adapted and the data source remains unchanged.

Quality control: SGML documents are parsed, which means they are checked for compliance with the regulations of the DTD. Deviations will result in error messages. In the example above, the sequence “first name, last name”

<first-name>Charles F.</first-name><last-name>Goldfarb</last-name>

is not allowed. In documents with strictly fixed structures, such as encyclopedias, the DTD can be used for automated quality control. In texts with relatively free structure, the DTD is formulated accordingly.

In many sectors of information processing, these advantages were recognized a long time ago and have been used since then. Company- and industry-wide, DTDs have been established for particular applications in order to optimize the interchange of homogenous information. For example in the automotive (J2008) and aviation industries (ATA 100, AECMA), in the publishing (ISO 12083, Majour) or software documentation sectors (DocBook). HTML is one of those DTDs. It was developed for presenting and processing information for the Web.

Next Step: XML, the Web Revolution

XML (eXtensible Markup Language), a recommendation of the World Wide Web Consortium, is a subject of SGML. XML is a meta-language (a form of language used to discuss language), which enables a general availability and interchange of information that is structured according to its content, with any kind of application, in various presentation, for different target groups and different purposes.

Differences between SGML and XML

The differences between SGML and XML arose from the aim to develop a meta-language especially for the needs of the Web and to promote a fast establishment of this language on the Web.

Simple implementation: The language capacity of XML is limited, and therefore the development of applications for XML is less complex than for SGML. The dissemination of XML for Web publishing is thus favored.

DTD not necessary: XML documents can be used without a DTD (Document Type Definition). Thus, XML can be used for structuring as regards content and as a pure presentation tool. It is possible to use it according to its purpose. For presentation and

downloading data, the document can be used separately in order to facilitate and accelerate processing. If the structure of the document is relevant and has to be controllable, then the document is transported together with its DTD.

If the DTD is missing, XML documents have to be well formed, that is their structure has to fulfill specific preconditions in order to be able to be interpreted and processed correctly in all applications.

1 Introduction to XML

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

2 Documents

A data object is an XML document if it is well formed, as defined in this specification. A well-formed XML document may in addition be valid if it meets certain further constraints.

Each XML document has both a logical and a physical structure. Physically, the document is composed of units called entities. An entity may refer to other entities to cause their inclusion in the document. A document begins in a "root" or document entity. Logically, the document is composed of declarations, elements, comments, character references, and processing instructions, all of which are indicated in the document by explicit markup. The logical and physical structures must nest properly.

Well-Formed XML Documents

A textual object is a well-formed XML document if:

- ✓ Taken as a whole, it matches the production labeled document.
- ✓ It meets all the well-formedness constraints given in this specification.
- ✓ Each of the parsed entities which is referenced directly or indirectly within the document is well-formed.

Document

Matching the document production implies that:

- It contains one or more elements.

- There is exactly one element, called the root, or document element, no part of which appears in the content of any other element. For all other elements, if the start-tag is in the content of another element, the end-tag is in the content of the same element. More simply stated, the elements, delimited by start- and end-tags, nest properly within each other.

As a consequence of this, for each non-root element C in the document, there is one other element P in the document such that C is in the content of P. C is not in the content of any other element that is in the content of P. P is referred to as the **parent** of C, and C as a **child** of P.

Characters

A parsed entity contains text, a sequence of characters, which may represent markup or character data. A character is an atomic unit of text as specified by ISO/IEC 10646. Legal characters are tab, carriage return, line feed, and the legal characters of Unicode and ISO/IEC 10646. New characters may be added to these standards by amendments or new editions. Consequently, XML processors must accept any character in the range specified for Char.

Character Range

The mechanism for encoding character code points into bit patterns may vary from entity to entity. All XML processors must accept the UTF-8 and UTF-16 encoding of 10646; the mechanisms for signaling which of the two is in use, or for bringing other encoding into play, are discussed, in Character Encoding in Entities.

Common Syntactic Constructs

This section defines some symbols used widely in the grammar.

S (white space) consists of one or more space characters, carriage returns, line feeds, or tabs.

White Space

Characters are classified for convenience as letters, digits, or other characters. A letter consists of an alphabetic or syllabic base character or an ideographic character. Full definitions of the specific characters in each class are given in B Character Classes.

A Name is a token beginning with a letter or one of a few punctuation characters, and continuing with letters, digits, hyphens, underscores, colons, or full stops, together known as name characters. Names beginning with the string "xml", or any string which would match (('X'|'x') ('M'|'m') ('L'|'l')), are reserved for standardization in this or future versions of this specification.

Note:

The Namespaces in XML Recommendation [XML Names] assigns a meaning to names containing colon characters. Therefore, authors should not use the colon in XML names except for namespace purposes, but XML processors must accept the colon as a name character.

A Nmtoken (name token) is any mixture of name characters.

Names and Tokens

NameChar
Name
Names
Nmtoken
Nmtokens

Literal data is any quoted string not containing the quotation mark used as a delimiter for that string. Literals are used for specifying the content of internal entities (EntityValue), the values of attributes (AttValue), and external identifiers (SystemLiteral). Note that a SystemLiteral can be parsed without scanning for markup.

Literals

EntityValue
AttValue
SystemLiteral
PubidLiteral
PubidChar

Note:

Although the EntityValue production allows the definition of an entity consisting of a single explicit < in the literal (e.g., <!ENTITY mylt "<">), it is strongly advised to avoid this practice since any reference to that entity will cause a well-formedness error.

Character Data and Markup

Text consists of intermingled character data and markup. Markup takes the form of start-tags, end-tags, empty-element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, processing instructions, XML declarations, text declarations, and any white space that is at the top level of the document entity (that is, outside the document element and not inside any other markup).

The ampersand character (&) and the left angle bracket (<) may appear in their literal form only when used as markup delimiters, or within a comment, a processing instruction, or a CDATA section.

In the content of elements, character data is any string of characters, which does not contain the start-delimiter of any markup. In a CDATA section, character data is any string of characters not including the CDATA-section-close delimiter.

To allow attribute values to contain both single and double quotes, the apostrophe or single-quote character (') may be represented as "'", and the double-quote character (") as """.

Comments

Comments may appear anywhere in a document outside other markup; in addition, they may appear within the document type declaration at places allowed by the grammar. They are not part of the document's character data; an XML processor may make it possible for an application to retrieve the text of comments. For compatibility, the string "--" (double-hyphen) must not occur within comments. Parameter entity references are not recognized within comments.

An example of a comment:

```
<!--declarations for <head> & <body> -->
```

Note that the grammar does not allow a comment ending in `--->`. The following example is *not* well formed.

```
<!-- B+, B, or B--->
```

Processing Instructions

PI
PITarget

Processing instructions (PIs) allow documents to contain instructions for applications. PIs are not part of the document's character data, but must be passed through to the application. The PI begins with a target (PITarget) used to identify the application to which the instruction is directed. The target names "XML", "xml", and so on are reserved for standardization in this or future versions of this specification. The XML Notation mechanism may be used for formal declaration of PI targets. Parameter entity references are not recognized within processing instructions.

CDATA Sections

CDATA sections may occur anywhere character data may occur; they are used to escape blocks of text containing characters, which would otherwise be recognized as markup. CDATA sections begin with the string "<![CDATA[" and end with the string "]]>".

CDSect
CDStart
CData
CDEnd

Within a CDATA section, only the CDEnd string is recognized as markup, so that left angle brackets and ampersands may occur in their literal form; they need not (and cannot) be escaped using "<" and "&". CDATA sections cannot nest.

An example of a CDATA section, in which "<greeting>" and "</greeting>" are recognized as character data, not markup:

```
<![CDATA[<greeting>Hello, world!</greeting>]]>
```

Prolog and Document Type Declaration

XML documents should begin with an XML declaration, which specifies the version of XML being used. For example, the following is a complete XML document, well-formed but not valid:

```
<?xml version="1.0"?> <greeting>Hello, world!</greeting>
```

and so is this:

```
<greeting>Hello, world!</greeting>
```

The version number "1.0" should be used to indicate conformance to this version of this specification; it is an error for a document to use the value "1.0" if it does not conform to this version of this specification. It is the intent of the XML working group to give later versions of this specification numbers other than "1.0", but this intent does not indicate a commitment to produce any future versions of XML, nor if any are produced, to use any particular numbering scheme. Since future versions are not ruled out, this construct is provided as a means to allow the possibility of automatic version recognition, should it become necessary. Processors may signal an error if they receive documents labeled with versions they do not support.

The function of the markup in an XML document is to describe its storage and logical structure and to associate attribute-value pairs with its logical structures. XML provides a mechanism, the document type declaration, to define constraints on the logical structure and to support the use of predefined storage units. It is important to refer that an XML document is valid if it has an associated document type declaration and if the document complies with the constraints expressed in it. The document type declaration must appear before the first element in the document.

The XML document type declaration contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as **a document type definition**, or **DTD**. The document type declaration can point to an external subset (a special kind of external entity) containing markup declarations, or can contain the markup declarations directly in an internal subset, or can do both. The DTD for a document consists of both subsets taken together.

A markup declaration is an element type declaration, an attribute-list declaration, an entity declaration, or a notation declaration. These declarations may be contained in whole or in part within parameter entities.

Document Type Definition

Note that it is possible to construct a well-formed document containing a doctypedecl that neither points to an external subset nor contains an internal subset.

The markup declarations may be made up in whole or in part of the replacement text of parameter entities. The productions later in this specification for individual nonterminals (elementdecl, AttlistDecl, and so on) describe the declarations after all the parameter entities have been included.

Parameter entity references are recognized anywhere in the DTD (internal and external subsets and external parameter entities), except in literals, processing instructions, comments, and the contents of ignored conditional sections. They are also recognized in entity value literals. The use of parameter entities in the internal subset is restricted as described below.

Validity constraint: Root Element Type

The Name in the document type declaration must match the element type of the root element.

Validity constraint: Proper Declaration/PE Nesting

Parameter-entity replacement text must be properly nested with markup declarations. That is to say, if either the first character or the last character of a markup declaration (markupdecl above) is contained in the replacement text for a parameter-entity reference, both must be contained in the same replacement text.

Well-formedness constraint: PEs in Internal Subset

In the internal DTD subset, parameter-entity references can occur only where markup declarations can occur, not within markup declarations. (This does not apply to references that occur in external parameter entities or to the external subset.)

Well-formedness constraint: External Subset

The external subset, if any, must match the production for extSubset.

Well-formedness constraint: PE between Declarations

The replacement text of a parameter entity reference in a DeclSep must match the production extSubsetDecl.

Like the internal subset, the external subset and any external parameter entities must consist of a series of complete markup declarations of the types allowed by the non-terminal symbol markup declaration, interspersed with white space or parameter-entity references. However, portions of the contents of the external subset or of these external parameter entities may conditionally be ignored by using the conditional section construct; this is not allowed in the internal subset.

External Subset

The external subset and external parameter entities also differ from the internal subset in that in them, parameter-entity references are permitted *within* markup declarations, not only *between* markup declarations.

An example of an XML document with a document type declaration:

```
<?xml version="1.0"?> <!DOCTYPE greeting SYSTEM "hello.dtd">
```

```
<greeting>Hello, world!</greeting>
```

The system identifier "hello.dtd" gives the address of a DTD for the document.

The declarations can also be given locally, as in this example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
<greeting>Hello, world!</greeting>
```

If both the external and internal subsets are used, the internal subset is considered to occur before the external subset. This has the effect that entity and attribute-list declarations in the internal subset take precedence over those in the external subset.

Standalone Document Declaration

Markup declarations can affect the content of the document, as passed from an XML processor to an application; examples are attribute defaults and entity declarations. The standalone document declaration, which may appear as a component of the XML declaration, signals whether or not there are such declarations, which appear external to the document entity or in parameter entities. Note that an external markup declaration is defined as a markup declaration occurring in the external subset or in a parameter entity (external or internal, the latter being included because non-validating processors are not required to read them).

Standalone Document Declaration

In a standalone document declaration, the value "yes" indicates that there are no external markup declarations, which affect the information passed from the XML processor to the application. The value "no" indicates that there are or may be such external markup declarations. Note that the standalone document declaration only denotes the presence of external declarations; the presence, in a document, of references to external entities, when those entities are internally declared, does not change its standalone status.

If there are no external markup declarations, the standalone document declaration has no meaning. If there are external markup declarations but there is no standalone document declaration, the value "no" is assumed.

Any XML document for which standalone = "no" holds can be converted algorithmically to a standalone document, which may be desirable for some network delivery applications.

Validity constraint: Standalone Document Declaration

The standalone document declaration must have the value "no" if any external markup declarations contain declarations of:

- ✓ attributes with default values, if elements to which these attributes apply appear in the document without specifications of values for these attributes, or
- ✓ entities (other than amp, lt, gt, apos, quot), if references to those entities appear in the document, or
- ✓ attributes with values subject to normalization, where the attribute appears in the document with a value which will change as a result of normalization, or
- ✓ element types with element content, if white space occurs directly within any instance of those types.

An example XML declaration with a standalone document declaration:

```
<?xml version="1.0" standalone='yes'?>
```

White Space Handling

In editing XML documents, it is often convenient to use "white space" (spaces, tabs, and blank lines) to set apart the markup for greater readability. Such white space is typically not intended for inclusion in the delivered version of the document. On the other hand, "significant" white space that should be preserved in the delivered version is common, for example in poetry and source code.

An XML processor must always pass all characters in a document that is not markup through to the application. A validating XML processor must also inform the application, which of these characters constitute white space appearing in element content.

A special attribute named `xml:space` may be attached to an element to signal an intention that in that element, white space should be preserved by applications. In valid documents, this attribute, like any other, must be declared if it is used. When declared, it must be given as an enumerated type whose values are one or both of "default" and "preserve". For example:

```
<!ATTLIST poem  xml:space (default|preserve) 'preserve'>
<!-- -->
<!ATTLIST pre xml:space (preserve) #FIXED 'preserve'>
```

The value "*default*" signals that applications' default white-space processing modes are acceptable for this element; the value "*preserve*" indicates the intent that

applications preserve all the white space. This declared intent is considered to apply to all elements within the content of the element where it is specified, unless overrides with another instance of the `xml:space attribute`.

The root element of any document is considered to have signaled no intentions as regards application space handling, unless it provides a value for this attribute or the attribute is declared with a default value.

End-of-Line Handling

XML parsed entities are often stored in computer files which, for editing convenience, are organized into lines. These lines are typically separated by some combination of the characters carriage-return (#xD) and line-feed (#xA).

To simplify the tasks of applications, the characters passed to an application by the XML processor must be as if the XML processor normalized all line breaks in external parsed entities (including the document entity) on input, before parsing, by translating both the two-character sequence #xD #xA and any #xD that is not followed by #xA to a single #xA character.

Language Identification

In document processing, it is often useful to identify the natural or formal language in which the content is written. A special attribute named `xml:lang` may be inserted in documents to specify the language used in the contents and attribute values of any element in an XML document. In valid documents, this attribute, like any other, must be declared if it is used. The values of the attribute are language identifiers, Tags for the Identification of Languages, or its successor on the IETF Standards Track.

For example:

```
<p xml:lang="en">The quick brown fox jumps over the lazy dog.</p>
<p xml:lang="en-GB">What colour is it?</p>
<p xml:lang="en-US">What color is it?</p>
<sp who="Faust" desc='leise' xml:lang="de">
  <l>Habe nun, ach! Philosophie,</l>
  <l>Juristerei, und Medizin</l>
  <l>und leider auch Theologie</l>
  <l>durchaus studiert mit heißem Bemüh'n.</l>
</sp>
```

The intent declared with `xml:lang` is considered to apply to all attributes and content of the element where it is specified, unless overridden with an instance of `xml:lang` on another element within that content.
A simple declaration for `xml:lang` might take the form

```
xml:lang NMTOKEN #IMPLIED
```

But specific default values may also be given, if appropriate. In a collection of French poems for English students, with glosses and notes in English, the `xml:lang` attribute might be declared this way:

```
<!ATTLIST poem   xml:lang NMTOKEN 'fr'>
<!ATTLIST gloss  xml:lang NMTOKEN 'en'>
<!ATTLIST note   xml:lang NMTOKEN 'en'>
```

3 Logical Structures

Each XML document contains one or more elements, the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by an empty-element tag. Each element has a type, identified by name, sometimes called its "generic identifier" (GI), and may have a set of attribute specifications. Each attribute specification has a name and a value.

Well-formedness constraint: Element Type Match

The Name in an element's end-tag must match the element type in the start-tag.

Validity constraint: Element Valid

An element is valid if there is a declaration matching where the Name matches the element type, and one of the following holds:

- ✓ The declaration matches EMPTY and the element has no content.
- ✓ The declaration matches children and the sequence of child elements belongs to the language generated by the regular expression in the content model, with optional white space between the start-tag and the first child element, between child elements, or between the last child element and the end-tag.
- ✓ The declaration matches Mixed and the content consist of character data and child elements whose types match names in the content model.
- ✓ The declaration matches ANY, and the types of any child elements have been declared.

Start-Tags, End-Tags, and Empty-Element Tags

The beginning of every non-empty XML element is marked by a start-tag.

Start-Tag

The Name in the start- and end-tags gives the element's type. The Name-AttValue pairs are referred to as the attribute specifications of the element, with the Name in each pair referred to as the attribute name and the content of the AttValue (the text between the ' or " delimiters) as the attribute value. Note that the order of attribute specifications in a start-tag or empty-element tag is not significant.

Well-formedness constraint:

No attribute name may appear more than once in the same start-tag or empty-element tag.

Validity constraint: Attribute Value Type

The attribute must have been declared; the value must be of the type declared for it.

Well-formedness constraint: No External Entity References

Attribute values cannot contain direct or indirect entity references to external entities.

Well-formedness constraint: No < in Attribute Values

The replacement text of any entity referred to directly or indirectly in an attribute value must not contain a <.

An example of a start-tag:

```
<termdef id="dt-dog" term="dog">
```

The end of every element that begins with a start-tag must be marked by an **end-tag** containing a name that echoes the element type as given in the start-tag.
End-tag

An example of an end-tag:

```
</termdef>
```

The text between the start-tag and end-tag is called the elements content.

Content of Elements

An element with no content is said to be empty. The representation of an empty element is either a start-tag immediately followed by an end-tag, or an empty-element tag. Note that an empty-element tag takes a special form.

Tags for Empty Elements

Empty-element tags may be used for any element, which has no content, whether or not it is declared using the keyword EMPTY. For interoperability, the empty-element tag should be only used, for elements, which are, declared EMPTY.

Examples of empty elements:

```
<IMG align="left"
 src="http://www.w3.org/Icons/WWW/w3c_home" />
<br></br>
<br/>
```

Element Type Declarations

The element structure of an XML document may, for validation purposes, be constrained using element type and attribute-list declarations. An element type declaration constrains the element's content.

Element type declarations often constrain which element types can appear as children of the element. At user option, an XML processor may issue a warning when a declaration mentions an element type for which no declaration is provided, but this is not an error.

Validity constraint: Unique Element Type Declaration

No element type may be declared more than once.

Examples of element type declarations:

```
<!ELEMENT br EMPTY>
<!ELEMENT p (#PCDATA|emph)* >
<!ELEMENT %name.para; %content.para; >
<!ELEMENT container ANY>
```

Element Content

An element type has element content when elements of that type must contain only child elements (no character data), optionally separated by white space. In this case, the constraint includes a content model, a simple grammar governing the allowed types of the child elements and the order in which they are allowed to appear. The grammar is built on content particles (cps), which consist of names, choice lists of content particles, or sequence lists of content particles:

Element-content Models

Any content particle in a choice list may appear in the element content at the location where the choice list appears in the grammar; content particles occurring in a sequence list must each appear in the element content in the order given in the list. The optional character governs whether the element or the content particles in the list may occur one or more (+), zero or more (*), or zero or one times (?). The absence of such an operator means that the element or content particle must appear exactly once. This syntax and meaning are identical to those used in the productions in this specification.

The content of an element matches a content model if and only if it is possible to trace out a path through the content model, obeying the sequence, choice, and repetition operators and matching each element in the content against an element type in the content model. For compatibility, it is an error if an element in the document can match more than one occurrence of an element type in the content model.

Validity constraint: Proper Group/PE Nesting

Parameter-entity replacement text must be properly nested with parenthesized groups. That is to say, if either of the opening or closing parentheses in a choice, seq, or Mixed construct is contained in the replacement text for a parameter entity, both must be contained in the same replacement text.

For interoperability, if a parameter-entity reference appears in a choice, seq, or Mixed construct, its replacement text should contain at least one non-blank character, and neither the first nor last non-blank character of the replacement text should be a connector (| or ,).

Examples of element-content models:

```
<!ELEMENT spec (front, body, back?)>
<!ELEMENT div1 (head, (p | list | note)*, div2*)>
<!ELEMENT dictionary-body (%div.mix; | %dict.mix;)*>
```

Mixed Content

An element type has mixed content when elements of that type may contain character data, optionally interspersed with child elements. In this case, the types of the child elements may be constrained, but not their order or their number of occurrences:

Mixed-content Declaration

Where the Names give the types of elements that may appear as children.

Validity constraint: No Duplicate Types

The same name must not appear more than once in a single mixed-content declaration. Examples of mixed content declarations:

```
<!ELEMENT p (#PCDATA|a|ul|b|I|em)*>
<!ELEMENT p (#PCDATA | %font; | %phrase; | %special; | %form;)* >
<!ELEMENT b (#PCDATA)>
```

Attribute-List Declarations

Attributes are used to associate name-value pairs with elements. Attribute specifications may appear only within start-tags and empty-element tags; thus, the productions used to recognize them appear in Start-Tags, End-Tags, and Empty-Element Tags. Attribute-list declarations may be used:

- To define the set of attributes pertaining to a given element type.
- To establish type constraints for these attributes.
- To provide default values for attributes.

Attribute-list declarations specify the name, data type, and default value (if any) of each attribute associated with a given element type.

Attribute-list Declaration

The AttlistDecl describes the type of an element. At user option, an XML processor may issue a warning if attributes are declared for an element type not itself declared, but this is not an error. The Name in the AttDef rule is the name of the attribute.

When more than one AttlistDecl is provided for a given element type, the contents of all those provided is merged. When more than one definition is provided for the same attribute of a given element type, the first declaration is binding and later declarations are ignored. For interoperability, writers of DTDs may choose to provide at most one attribute-list declaration for a given element type, at most one attribute definition for a given attribute name in an attribute-list declaration, and at least one attribute definition in each attribute-list declaration. For interoperability, an XML processor may at user option issue a warning when more than one attribute-list declaration is provided for a given element type, or more than one attribute definition is provided for a given attribute, but this is not an error.

Attribute Types

XML attribute types are of three kinds: a string type, a set of tokenized types, and enumerated types. The string type may take any literal string as a value; the tokenized types have varying lexical and semantic constraints. The validity constraints noted in the grammar are applied after the attribute value has been normalized as described in Attribute-List Declarations.

Attribute Types

Validity constraint: ID
Values of type ID must match the Name production. A name must not appear more than once in an XML document as a value of this type; i.e., ID values must uniquely identify the elements, which bear them.

Validity constraint: One ID per Element Type

No element type may have more than one ID attribute specified.

Validity constraint: ID Attribute Default

An ID attribute must have a declared default of #IMPLIED or #REQUIRED.

Validity constraint: IDREF

Values of type IDREF must match the Name production, and values of type IDREFS must match Names; each Name must match the value of an ID attribute on some element in the XML document; i.e. IDREF values must match the value of some ID attribute.

Validity constraint: Name Token

Values of type NMTOKEN must match the Nmtoken production; values of type NMTOKENS must match Nmtokens.

Enumerated attributes can take one of a list of values provided in the declaration. There are two kinds of enumerated types.

Enumerated Attribute Types

A **NOTATION** attribute identifies a notation, declared in the DTD with associated system and/or public identifiers, to be used in interpreting the element to which the attribute is attached.

Validity constraint: Notation Attributes

Values of this type must match one of the notation names included in the declaration; all notation names in the declaration must be declared.

Validity constraint: One Notation Per Element Type

No element type may have more than one NOTATION attribute specified.

Validity constraint: No Notation on Empty Element

For compatibility, an attribute of type NOTATION must not be declared on an element declared EMPTY.

Validity constraint: Enumeration

Values of this type must match one of the Nmtoken tokens in the declaration.

For interoperability, the same Nmtoken should not occur more than once in the enumerated attribute types of a single element type.

Attribute Defaults

An attribute declaration provides information on whether the attribute's presence is required, and if not, how an XML processor should react if a declared attribute is absent in a document.

Attribute Defaults

In an attribute declaration, #REQUIRED means that the attribute must always be provided, #IMPLIED that no default value is provided. If the declaration is neither #REQUIRED nor #IMPLIED, then the AttValue value contains the declared default value; the #FIXED keyword states that the attribute must always have the default value.

If a default value is declared, when an XML processor encounters an omitted attribute, it is to behave as though the attribute was present with the declared default value.

Validity constraint: Required Attribute

If the default declaration is the keyword #REQUIRED, then the attribute must be specified for all elements of the type in the attribute-list declaration.

Validity constraint: Attribute Default Legal

The declared default value must meet the lexical constraints of the declared attribute type.

Validity constraint: Fixed Attribute Default

If an attribute has a default value declared with the #FIXED keyword, instances of that attribute must match the default value.

Examples of attribute-list declarations:

```
<!ATTLIST termdef
          id      ID      #REQUIRED
          name    CDATA   #IMPLIED>
<!ATTLIST list
          type    (bullets|ordered|glossary)  "ordered">
<!ATTLIST form
          method  CDATA   #FIXED "POST">
```

Attribute-Value Normalization

Before the value of an attribute is passed to the application or checked for validity, the XML processor must normalize the attribute value by applying the algorithm below, or by using some other method such that the value passed to the application is the same as that produced by the algorithm.

1. All line breaks must have been normalized on input to #xA as described in 2.11 End-of-Line Handling, so the rest of this algorithm operates on text normalized in this way.

2. Begin with a normalized value consisting of the empty string.

    - For each character, entity reference, or character reference in the unnormalized attribute value, beginning with the first and continuing to the last, do the following:

    - For a character reference, append the referenced character to the normalized value.

- For an entity reference, recursively apply step 3 of this algorithm to the replacement text of the entity.

- For another character, append the character to the normalized value.

If the attribute type is not CDATA, then the XML processor must further process the normalized attribute value by discarding any leading and trailing space characters, and by replacing sequences of space characters by a single space character.

Note that if the unnormalized attribute value contains a character reference to a white space character other than space the normalized value contains the referenced character itself. This contrasts with the case where the unnormalized value contains a white space character (not a reference), which is replaced with a space character in the normalized value and also contrasts with the case where the unnormalized value contains an entity reference whose replacement text contains a white space character; being recursively processed, the white space character is replaced with a space character in the normalized value.

All attributes for which no declaration has been read, should be treated by a non-validating processor as if declared CDATA.

Following are examples of attribute normalization. Given the following declarations:

```
<!ENTITY d "&#xD;">
<!ENTITY a "&#xA;">
<!ENTITY da "&#xD;&#xA;">
```

The attribute specifications in the left column below would be normalized to the character sequences of the middle column if the attribute a is declared NMTOKENS and to those of the right columns if a is declared CDATA.

Attribute specification	a is NMTOKENS	a is CDATA
a=" xyz"	x y z	#x20 #x20 x y z
a="&d;&d;A&a;&a;B&da;"	A #x20 B	#x20 #x20 A #x20 #x20 B #x20 #x20
a= "

A

B
 " | #xD #xD A #xA #xA B #xD #xA | #xD #xD A #xA #xA B #xD #xD |

Note that the last example is invalid (but well formed) if a is declared to be of type **NMTOKENS**.

Conditional Sections

Conditional sections are portions of the document type declaration external subset which are included in, or excluded from, the logical structure of the DTD based on the keyword which governs them.

Validity constraint: Proper Conditional Section/PE Nesting

If any of the "<![", "[", or "]]>" of a conditional section is contained in the replacement text for a parameter-entity reference, all of them must be contained in the same replacement text.

Like the internal and external DTD subsets, a conditional section may contain one or more complete declarations, comments, processing instructions, or nested conditional sections, intermingled with white space.

If the keyword of the conditional section is **INCLUDE**, then the contents of the conditional section are part of the DTD. If the keyword of the conditional section is **IGNORE**, then the contents of the conditional section are not logically part of the DTD. If a conditional section with a keyword of **INCLUDE** occurs within a larger conditional section with a keyword of **IGNORE**, both the outer and the inner conditional sections are ignored. The contents of an ignored conditional section are parsed by ignoring all characters after the "[" following the keyword, except conditional section starts "<![" and ends "]]>", until the matching conditional section end is found. Parameter entity references are not recognized in this process.

If the keyword of the conditional section is a parameter-entity reference, the parameter entity must be replaced by its content before the processor decides whether to include or ignore the conditional section.

An example:

```
<!ENTITY % draft 'INCLUDE' >
<!ENTITY % final 'IGNORE' >

<![%draft;[
<!ELEMENT book (comments*, title, body, supplements?)>
]]>
<![%final;[
<!ELEMENT book (title, body, supplements?)>
]]>
```

4 Physical Structures

An XML document may consist of one or many storage units. These are called entities; they all have content and are all (except for the document entity and the external DTD subset) identified by entity name. Each XML document has one entity called the document entity, which serves as the starting point for the XML processor and may contain the whole document.

Entities may be either parsed or unparsed. A parsed entity's contents are referred to as its replacement text; this text is considered an integral part of the document.

An unparsed entity is a resource whose contents may or may not be text, and if text, may be other than XML. Each unparsed entity has an associated notation, identified by name. Beyond a requirement that an XML processor make the identifiers for the entity and notation available to the application, XML places no constraints on the contents of unparsed entities.

Parsed entities are invoked by name using entity references; unparsed entities by name, given in the value of ENTITY or ENTITIES attributes.

General entities are entities for use within the document content. In this specification, general entities are sometimes referred to with the unqualified term entity when this leads to no ambiguity. Parameter entities are parsed entities for use within the DTD. These two types of entities use different forms of reference and are recognized in different contexts. Furthermore, they occupy different namespaces; a parameter entity and a general entity with the same name are two distinct entities.

Character and Entity References

A character reference refers to a specific character in the ISO/IEC 10646 character set, for example one not directly accessible from available input devices.

Well-formedness constraint: Legal Character

Characters referred to using character references must match the production for Char.

If the character reference begins with "&#x", the digits and letters up to the terminating; provide a hexadecimal representation of the character's code point in ISO/IEC 10646. If it begins just with "&#", the digits up to the terminating; provide a decimal representation of the character's code point.

An entity reference refers to the content of a named entity. References to parsed general entities use ampersand (&) and semicolon (;) as delimiters. Parameter-entity references use percent-sign (%) and semicolon (;) as delimiters.

Entity Reference

Well-formedness constraint: Entity Declared

In a document without any DTD, (a document with only an internal DTD subset, which contains no parameter entity references, or a document with "standalone='yes'", for an entity reference that does not occur within the external subset or a parameter entity), the Name given in the entity reference must match that in an entity declaration that does not occur within the external subset or a parameter entity, except that well-formed documents need not declare any of the following entities: amp, lt, gt, apos, quot. The declaration of a general entity must precede any reference to it, which appears in a default value in an attribute-list declaration.

Note that if entities are declared in the external subset or in external parameter entities, a non-validating processor is not obligated to read and process their declarations; for such documents, the rule that an entity must be declared is a well-formedness constraint only if standalone='yes'.

Validity constraint: Entity Declared

In a document with an external subset or external parameter entities with "standalone='no'", the Name given in the entity reference must match that in an entity declaration. For interoperability, valid documents should declare the entities amp, lt, gt, apos, and quot, in the form specified in Predefined Entities. The declaration of a parameter entity must precede any reference to it. Similarly, the declaration of a general entity must precede any attribute-list declaration containing a default value with a direct or indirect reference to that general entity.

Well-formedness constraint: Parsed Entity

An entity reference must not contain the name of an unparsed entity. Unparsed entities may be referred to only in attribute values declared to be of type ENTITY or ENTITIES.

Well-formedness constraint: No Recursion

A parsed entity must not contain a recursive reference to itself, either directly or indirectly.

Well-formedness constraint: In DTD

Parameter-entity references may only appear in the DTD.

Examples of character and entity references:

```
Type <key>less-than</key> (&#x3C;) to save options.
This document was prepared on &docdate; and
is classified &security-level;.
```

Example of a parameter-entity reference:

```
<!-- declare the parameter entity "ISOLat2"... -->
```

```
<!ENTITY % ISOLat2
         SYSTEM "http://www.xml.com/iso/isolat2-xml.entities" >
<!-- ... now reference it. -->
%ISOLat2;
```

Entity Declarations

Entities are declared thus:

The Name identifies the entity in an entity reference or, in the case of an unparsed entity, in the value of an ENTITY or ENTITIES attribute. If the same entity is declared more than once, the first declaration encountered is binding; at user option, an XML processor may issue a warning if entities are declared multiple times.

Internal Entities

[Definition: If the entity definition is an EntityValue, the defined entity is called an internal entity. There is no separate physical storage object, and the content of the entity is given in the declaration.] Note that some processing of entity and character references in the literal entity value may be required to produce the correct replacement text: see 4.5 Construction of Internal Entity Replacement Text.

An internal entity is a parsed entity.

Example of an internal entity declaration:

```
<!ENTITY Pub-Status "This is a pre-release of the
 specification.">
```

External Entities

If the entity is not internal, it is an external entity, declared as follows:

External Entity Declaration

Validity constraint: Notation Declared

The Name must match the declared name of a notation.

The SystemLiteral is called the entity's system identifier. It is a URI reference (as defined in, updated by , meant to be dereferenced to obtain input for the XML processor to construct the entity's replacement text. It is an error for a fragment identifier (beginning with a # character) to be part of a system identifier. Unless otherwise provided by information outside the scope of this specification (e.g. a special XML element type defined by a particular DTD, or a processing instruction defined by a particular application specification), relative URIs are relative to the location of the resource within which the entity declaration occurs. A URI might thus be relative to the document entity, to the entity containing the external DTD subset, or to some other external parameter entity.

URI references require encoding and escaping of certain characters. The disallowed characters include all non-ASCII characters, plus the excluded characters, except for the number sign (#) and percent sign (%) characters and the square bracket characters re-allowed in . Disallowed characters must be escaped as follows:

- Each disallowed character is converted to UTF-8 as one or more bytes.
- Any octets corresponding to a disallowed character are escaped with the URI escaping mechanism (that is, converted to %HH, where HH is the hexadecimal notation of the byte value).
- The original character is replaced by the resulting character sequence.

In addition to a system identifier, an external identifier may include a public identifier. An XML processor attempting to retrieve the entity's content may use the public identifier to try to generate an alternative URI reference. If the processor is unable to do so, it must use the URI reference specified in the system literal. Before a match is attempted, all strings of white space in the public identifier must be normalized to single space characters, and leading and trailing white space must be removed.

Examples of external entity declarations:

```
<!ENTITY open-hatch
         SYSTEM "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY open-hatch
         PUBLIC "-//Textuality//TEXT Standard open-hatch
boilerplate//EN"
         "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY hatch-pic
         SYSTEM "../grafix/OpenHatch.gif"
         NDATA gif >
```

Parsed Entities

The Text Declaration

External parsed entities should each begin with a text declaration.

Text Declaration

The text declaration must be provided literally, not by reference to a parsed entity. No text declaration may appear at any position other than the beginning of an external parsed entity. The text declaration in an external parsed entity is not considered part of its replacement text.

Well-Formed Parsed Entities

The document entity is well formed if it matches the production labeled document. An external general parsed entity is well formed if it matches the production labeled extParsedEnt. All external parameter entities are well formed by definition.

Well-Formed External Parsed Entity

An internal general parsed entity is well formed if its replacement text matches the production labeled content. All internal parameter entities are well formed by definition.

A consequence of well-formedness in entities is that the logical and physical structures in an XML document are properly nested; no start-tag, end-tag, empty-element tag, element, comment, processing instruction, character reference, or entity reference can begin in one entity and end in another.

Character Encoding in Entities

Each external parsed entity in an XML document may use a different encoding for its characters. All XML processors must be able to read entities in both the UTF-8 and UTF-16 encoding. The terms "UTF-8" and "UTF-16" in this specification do not apply to character encoding with any other labels, even if the encoding or labels are very similar to UTF-8 or UTF-16. XML processors must be able to use this character to differentiate between UTF-8 and UTF-16 encoded documents.

Although an XML processor is required to read only entities in the UTF-8 and UTF-16 encodings, it is recognized that other encodings are used around the world, and it may be desired for XML processors to read entities that use them. In the absence of external character encoding information (such as MIME headers), parsed entities which

are stored in an encoding other than UTF-8 or UTF-16 must begin with a text declaration containing an encoding declaration:

Encoding Declaration

They are errors:

- When an XML processor encounters an entity with an encoding that it is unable to process.
- If an XML entity is determined (via default, encoding declaration, or higher-level protocol) to be in a certain encoding but contains octet sequences that are not legal in that encoding.
- It is also a fatal error if an XML entity contains no encoding declaration and its content is not legal UTF-8 or UTF-16.

Examples of text declarations containing encoding declarations:

```
<?xml encoding='UTF-8'?>
<?xml encoding='EUC-JP'?>
```

XML Processor Treatment of Entities and References

The table below summarizes the contexts in which character references, entity references, and invocations of unparsed entities might appear and the required behavior of an XML processor in each case. The labels in the leftmost column describe the recognition context:

Reference in Content
As a reference anywhere after the start-tag and before the end-tag of an element; corresponds to the nonterminal content.

Reference in Attribute Value
As a reference within either the value of an attribute in a start-tag, or a default value in an attribute declaration; corresponds to the nonterminal AttValue.

Occurs as Attribute Value
As a Name, not a reference, appearing either as the value of an attribute, which has been declared as type ENTITY, or as one of the space-separated tokens in the value of an attribute, which has been declared as type ENTITIES.

Reference in Entity Value
As a reference within a parameter or internal entity's literal entity value in the entity's declaration; corresponds to the nonterminal EntityValue.

Reference in DTD
As a reference within either the internal or external subsets of the DTD, but outside of an EntityValue, AttValue, PI, Comment, SystemLiteral, PubidLiteral, or the contents of an ignored conditional section.

	Entity Type				Character
	Parameter	Internal General	External Parsed General	Unparsed	
Reference in Content	*Not recognized*	*Included*	*Included if validating*	*Forbidden*	*Included*
Reference in Attribute Value	*Not recognized*	*Included in literal*	*Forbidden*	*Forbidden*	*Included*
Occurs as Attribute Value	*Not recognized*	*Forbidden*	*Forbidden*	*Notify*	*Not recognized*
Reference in EntityValue	*Included in literal*	*Bypassed*	*Bypassed*	*Forbidden*	*Included*
Reference in DTD	*Included as PE*	*Forbidden*	*Forbidden*	*Forbidden*	*Forbidden*

Not Recognized
Outside the DTD, the % character has no special significance; thus, what would be parameter entity references in the DTD are not recognized as markup in content. Similarly, the names of unparsed entities are not recognized except when they appear in the value of an appropriately declared attribute.

Included
An entity is included when its replacement text is retrieved and processed, in place of the reference itself, as though it were part of the document at the location the reference was recognized. The replacement text may contain both character data and (except for parameter entities) markup, which must be recognized in the usual way. (The string "AT&T;" expands to "AT&T;" and the remaining ampersand is not recognized as an entity-reference delimiter.) A character reference is included when the indicated character is processed in place of the reference itself.

Included if validating
When an XML processor recognizes a reference to a parsed entity, in order to validate the document, the processor must include its replacement text. If the entity is external, and the processor is not attempting to validate the XML document, the processor may, but

need not, include the entity's replacement text. If a non-validating processor does not include the replacement text, it must inform the application that it recognized, but did not read, the entity.

This rule is based on the recognition that the automatic inclusion provided by the SGML and XML entity mechanism, primarily designed to support modularity in authoring, is not necessarily appropriate for other applications, in particular document browsing. Browsers, for example, when encountering an external parsed entity reference, might choose to provide a visual indication of the entity's presence and retrieve it for display only on demand.

Forbidden

The following are forbidden, and constitute fatal errors:

- The appearance of a reference to an unparsed entity.
- The appearance of any character or general-entity reference in the DTD except within an EntityValue or AttValue.
- A reference to an external entity in an attribute value.

Included in Literal

When an entity reference appears in an attribute value, or a parameter entity reference appears in a literal entity value, its replacement text is processed in place of the reference itself as though it were part of the document at the location the reference was recognized. For example, this is well formed:

```
<!--  -->
<!ENTITY % YN '"Yes"' >
<!ENTITY WhatHeSaid "He said %YN;" >
```

While this is not:

```
<!ENTITY EndAttr "27'" >
<element attribute='a-&EndAttr;>
```

Notify

When the name of an unparsed entity appears as a token in the value of an attribute of declared type ENTITY or ENTITIES, a validating processor must inform the application of the system and public (if any) identifiers for both the entity and its associated notation.

Bypassed
When a general entity reference appears in the EntityValue in an entity declaration, it is bypassed and left as is.

Included as PE
Just as with external parsed entities, parameter entities need only be included if validating. When a parameter-entity reference is recognized in the DTD and included, its replacement text is enlarged by the attachment of one leading and one following space (#x20) character; the intent is to constrain the replacement text of parameter entities to contain an integral number of grammatical tokens in the DTD. This behavior does not apply to parameter entity references within entity values.

Construction of Internal Entity Replacement Text

In discussing the treatment of internal entities, it is useful to distinguish two forms of the entity's value. The literal entity value is the quoted string actually present in the entity declaration, corresponding to the non-terminal EntityValue. The replacement text is the content of the entity, after replacement of character references and parameter-entity references.

The literal entity value as given in an internal entity declaration (EntityValue) may contain character, parameter-entity, and general-entity references. Such references must be contained entirely within the literal entity value. The actual replacement text that is included as described above must contain the replacement text of any parameter entities referred to, and must contain the character referred to, in place of any character references in the literal entity value; however, general-entity references must be left as-is, unexpanded. For example, given the following declarations:

```
<!ENTITY % pub    "&#xc9;ditions Gallimard" >
<!ENTITY   rights "All rights reserved" >
<!ENTITY   book   "La Peste: Albert Camus,
&#xA9; 1947 %pub;. &rights;" >
```

Then the replacement text for the entity "`book`" is:

```
La Peste: Albert Camus,
© 1947 Éditions Gallimard. &rights;
```

The general-entity reference "`&rights;`" would be expanded should the reference "`&book;`" appear in the document's content or an attribute value.

These simple rules may have complex interactions; for a detailed discussion of a difficult example, see D Expansion of Entity and Character References.

Predefined Entities

Entity and character references can both be used to escape the left angle bracket, ampersand, and other delimiters. A set of general entities (amp, lt, gt, apos, quot) is specified for this purpose. Numeric character references may also be used; they are expanded immediately when recognized and must be treated as character data, so the numeric character references "<" and "&" may be used to escape < and & when they occur in character data.

All XML processors must recognize these entities whether they are declared or not. For interoperability, valid XML documents should declare these entities, like any others, before using them. If the entities lt or amp are declared, they must be declared as internal entities whose replacement text is a character reference to the respective character (less-than sign or ampersand) being escaped; the double escaping is required for these entities so that references to them produce a well-formed result. If the entities gt, apos, or quot are declared, they must be declared as internal entities whose replacement text is the single character being escaped (or a character reference to that character; the double escaping here is unnecessary but harmless).

For example:

```
<!ENTITY lt     "&#38;#60;">
<!ENTITY gt     "&#62;">
<!ENTITY amp    "&#38;#38;">
<!ENTITY apos   "&#39;">
<!ENTITY quot   "&#34;">
```

Notation Declarations

Notations identify by name the format of unparsed entities, the format of elements, which bear a notation attribute, or the application to which a processing instruction is addressed.

Notation declarations provide a name for the notation, for use in entity and attribute-list declarations and in attribute specifications, and an external identifier for the notation which may allow an XML processor or its client application to locate a helper application capable of processing data in the given notation.

Notation Declarations

Validity constraint: Unique Notation Name

Only one notation declaration can declare a given Name.

XML processors must provide applications with the name and external identifier(s) of any notation declared and referred to an attribute value, attribute definition, or entity declaration. They may additionally resolve the external identifier into the system identifier, file name, or other information needed to allow the application to call a

processor for data in the notation described. (It is not an error, however, for XML documents to declare and refer to notations for which notation-specific applications are not available on the system where the XML processor or application is running.)

Document Entity

The document entity serves as the root of the entity tree and a starting-point for an XML processor. This specification does not specify how the document entity is to be located by an XML processor; unlike other entities, the document entity has no name and might well appear on a processor input stream without any identification at all.

5 Conformance

Validating and Non-Validating Processors

Conforming XML processors fall into two classes: validating and non-validating.

Validating and non-validating processors alike must report violations of this specification's well-formedness constraints in the content of the document entity and any other parsed entities that they read.

Validating processors must, at user option, report violations of the constraints expressed by the declarations in the DTD, and failures to fulfill the validity constraints given in this specification. To accomplish this, validating XML processors must read and process the entire DTD and all external parsed entities referenced in the document.

Non-validating processors are required to check only the document entity, including the entire internal DTD subset, for well formedness. While they are not required to check the document for validity, they are required to process all the declarations they read in the internal DTD subset and in any parameter entity, (that they read), up to the first reference to a parameter entity (that they do not read). That is to say, they must use the information in those declarations to normalize attribute values, include the replacement text of internal entities, and supply default attribute values. Except when standalone="yes", they must not process entity declarations or attribute-list declarations encountered after a reference to a parameter entity that is not read, since the entity may have contained overriding declarations.

Using XML Processors

The behavior of a validating XML processor is highly predictable; it must read every piece of a document and report all well formedness and validity violations. Less is required of a non-validating processor; it need not read any part of the document other than the document entity. This has two effects that may be important to users of XML processors:

- A non-validating processor may not detect certain well-formedness errors, specifically those that require reading external entities. Examples include the constraints entitled Entity Declared, Parsed Entity, and No Recursion, as well as some of the cases described as forbidden.

- The information passed from the processor to the application may vary, depending on whether the processor reads parameter and external entities. For example, a non-validating processor may not normalize attribute values, include the replacement text of internal entities, or supply default attribute values, where doing so depends on having read declarations in external or parameter entities.

For maximum reliability in interoperating between different XML processors, applications, which use non-validating processors, should not rely on any behaviors not required of such processors. Applications, which require facilities such as the use of default attributes, or internal entities, which are declared in external entities, should use validating XML processors.

6.Comparing XML and HTML

XML is not an improvement of HTML; it is a change in concept. HTML is a language. XML is a meta-language; a language to generate languages (any kind of language), perfectly suited to the respective purpose, for marking up information of any kind.

Flexibility instead of fixed structure: whereas HTML provides only a fixed amount of elements for heterogeneous information, XML is able to generate elements, which are tailor-made to particular information types. Usually, essential information is lost in Web publishing since HTML does not provide sufficient modes of expression. XML makes this information “Web-capable”. HTML tells how the data should look, but XML tells what it means.

Access to information instead of only to layout: HTML offers limited possibilities for structuring documents according to their content. It is mainly a layout-oriented language for the display of documents. Target retrieval, for example is not possible with HTML, but it is with XML and SGML data (e.g. search for the last name of an author).

Control: Just like in SGML, it is possible to parse documents. The XML DTD can be formulated in a way so that the basic requirements on the structure of the documents (e.g. sequence, existence of particular elements) are automatically verifiable (via an XML parser).

7. XML in the ADS

The Astrophysics Data System (ADS) provides access over 1.5 million references in Astronomy, Physics/Geophysics, and Space Instrumentation. These data come from many different sources in many different formats.

The ADS receives references and abstracts from many sources. This includes different versions of the same reference from several sources. In order to utilize this information efficiently, it is necessary to retain all the different versions of a reference, identify the origin of each reference item, and decide which version of a reference item is the best to show to users. XML provides an ideal framework for such a dataset. It allows the storage of several versions of the same item (for instance sets of keywords) that are tagged with their origin.

A second part of the upgrade concerns the use of extended character sets. Author names for instance use more than just the set of characters in the Latin-1 character set. In a discipline that uses mathematical symbols extensively, titles and abstracts frequently contain special characters like Greek letters and various other symbols. These can accurately be represented as Unicode characters. This will allow us to support any character in the original data.

Another aspect of the XML system will be to separate several of the data fields in more detail. For instance the journal field currently contains all the publisher information in one string. The XML version will eventually separate this field in several subfields like volume, issue, first page, last page, year, publisher, editor, etc. in a very consistent and straightforward manner. Once this XML system is fully implemented, it will allow both users and software tools more control over the formatting of the returned references.

8. Which language should be used for which purpose

The question whether XML or SGML is the right source format is not easily answered for all cases.

There is no question about existing SGML environments where highly functional applications are already available and the processing of SGML data is carried out effectively and is automated to a great extend. XML can be used as a further output format for Web publishing or data interchange. The conversion of the existing SGML data into the output format XML is generally simple.

XML is also suitable as a source format for simply structured documents and data that are only used for Web publishing and that cannot be encoded optimally with HTML. However, in order to take advantage of the entire range of XML's functionality, a detailed analysis of the data structure and purpose of use is required. Although it is not a requirement within XML to develop DTDs, it is absolutely recommended for professional data processing and maintenance.

It could be said in general that SGML offers the perfect basis for the entry and editing of deeply structured information as the entire language capacity can be used for data modeling

9. Related Languages and Derivatives

Being a meta-language, XML is more than a tool for semantic structuring of information in documents. With XML it is also possible to create languages which standardize the treatment of any kind of information, such as the presentation of data (style), data transfer, access of applications to the structure, description of the relationship between data and data modules (hyperlinks, addresses) or the communication between applications.

XDF, the eXtensible Data Format for Scientific Data

The following pages describe an XML mark-up language for documents containing most classes of scientific data. This language allows common data structures to be represented in a consistent manner independent of the scientific specialty involved. The language makes use of object models encompassed by modern programming languages. Such data representations would benefit from the widespread acceptance that XML has, and could bring about greater interdisciplinary information transfer. It is reasonable to expect that this approach would lead to a greater amount of clear public dissemination of scientific and technical explorations.

In XDF, data is described in a manner that accurately reflects scientists' views of data. Most scientific numerical data can be seen as an object assembled from objects of two categories:

1) A simple parameter that is set to a single value, possibly plus or minus infinity, or a range of values, possibly of infinite extent (Eg. x=3.1 or 0 < y < 180).

2) Gridded samples of scalar or vector fields embedded in an N-dimensional space (field arrays). These include tightly sampled grids such as spectra, images, animations, or time-series measurements. And, they include sparsely sampled fields such as interferometric u-v maps, event detection, or sets of pointed telescope observations.

Examples of N-dimensional spaces include physical space, projected space, time, wavelength, frequency, energy scales or some other parameter space. Complex numbers can be considered as vectors in the complex plane.

One often assembles these two basis objects into lists of lists or field arrays. A record is a list of values for a selection of parameters for a particular item or target. A list of these records is a table.

Because observations of data always have some finite resolution, it is always gridded (sometimes variably). Therefore both field arrays and tables can be represented similarly as ordered N-dimensional data cubes. Software can take advantage of this similarity by sharing input/output methods between the two. In fact much of the data handling can be similar: subsetting, taking cross sectioning, etc. However, the

fundamental difference between the two categories is the fact that tabled properties rarely form a continuous space and therefore interpolation and analyses depending on interpolation are not sensible.

For XDF, the field arrays and tables are considered to be a single class of N-dimensional objects, that can contain any combination of four distinct types of dimensions; continuous coordinates, discontinuous item and field spaces, and discontinuous vector components that usually refer to continuous coordinates.

A point of departure for the XDF format is a standardized format for associating the embedded space with the data and for keeping data associated with the same space together in a consistent and organized manner. This has been a failing of most of the existing scientific data formats probably because they are too flexible with associated scales and axes descriptions. It is often extremely difficult to create an application that can display axes along with all of the data without some user intervention. And too often the axes information is simply not present in the data file.

It is not intended for XDF to redefine all of the header data (metadata) that are associated with data. Each discipline should include into the XDF document its own elements for describing the circumstances of the data collection and relevant information for understanding the details. The advantages of XDF come primarily from having a standard core that directs the data reading and the relative positioning of multiple arrays. However, it should also be helpful to express the metadata of the older file formats in XML because it can then reside in well organized structures and benefit from standard interfaces for parsing and transformation.

The objects mentioned in the category 2 data class (tables), are objects in a broad sense. Sometimes they are simply things like stars, people, particles, etc. But, sometimes they are subsets of things like locations on a thing, for instance, shells in the interior of a star at particular depths, or locations on the surface of the Earth within a scan from a down looking satellite. Traditionally, the objects of tables are placed in the first column. For the human reader, this is fine because the first column can be easily scanned by eye. For machine-readable forms, it is preferable for the list of objects to become part of the metadata along with the field types. The reason for this is that queries are most often formulated for these quantities and it takes much longer to read all of the data then to read just the metadata. XDF allows for, but does not insists on, extracting object lists from the data files and placing them instead in the XDF document as axes. The document can link to the rest of the table, but not read unless the query result indicates that the data there is needed.

One of the key concepts in object oriented methodology is that data should be wrapped with the information necessary to read it and to make it useful. The XML language allows for this by including in data documents either references to applications or code in the form of ECMAscript or Java. An XML document can have references to files containing data and different types of data files can be handled by different applications. This not only allows input and preliminary processing to be self directed, but it also allows some of the data to be generated on the end users' machines. **XDF includes functions for calculating values along axes, and if necessary, calculating positional information for every grid point**. When applications are referenced or embedded within the data documents, it greatly reduces the learning curve needed to begin working with scientific data.

Specifics:

The XDF is a container for parameters, field arrays, and tables. Field arrays and tables are described by array elements. These contain axes information and data elements. Data elements are ordered lists of values of numeric or string types. The array and parameter elements can be grouped into structure elements. A simplified structure with an image looks like this:

```
<XDF>
  <structure>
    <array>
     <axis name="X-axis">
                <values> a list of values along one dimension</values>
       </axis>
     <axis name="Y-axis">
                <values> a list of values along other dimension</values>
       </axis>
       <read> info on the ordering of the data values and record
                        format.
       <recordFormat>...</recordFormat>
       </read>
     <data>
           The Data goes here
     </data>
    </array>

    <array>
      Some other array of data...
    </array>

  </structure>
</XDF>
```

The structure element is not necessary for this example, but it becomes useful when more complicated sets of arrays are involved.

AIML (Astronomical Instrument Markup Language)

AIML is an instrument description language that encompasses instrument characteristics, control commands, data stream descriptions (including image and housekeeping data), message formats, communication mechanisms, and pipeline algorithm descriptions. AIML also supports role-specific documentation and GUI component generation. AIML is a specific implementation of the more general Astronomical Markup Language.

AML (Astronomical Markup Language)

AML is an XML language describing various kinds of data useful in astronomy, intended to be an exchange formats for astronomical data, and especially metadata, over the Internet. The current DTD (December 1998) has 7 parts: Metadata, Astronomical Object, Article, Table, Set of Tables, Image, and Person. A Java applet can be used to browse AML documents, as easily as one would browse HTML documents, but with some features specific for astronomical data.

XSL (eXtensible Scientific Interchange Language)

XSL is a flexible, hierarchical, extensible, transport language for scientific data objects. The entire object may be represented in the file, or there may be metadata in the XSL file, with a powerful, fault-tolerant linking mechanism to external data. The language is based on XML, and is designed not only for parsing and processing by machines, but also for presentation to humans through web browsers and web-database technology.

3.CONVERSION OF EPN FORMAT TO XML FORMAT

As it has already been mentioned, in order to build XML applications, one needs to fulfill do four tasks:

- Select or write a DTD file
- Generate XML documents
- Interpret XML documents, and
- Display XML documents

A Document Type Definition (DTD) file defines what tags exists in a document, what tags contain other tags, the number and the sequence of the tags, the attributes the tags have, and optionally, the values those attributes have. DTD can be thought of as a template that should be filled. The eXtensible Markup Language (XML) file contains the tagged data.

In this thesis a converting program, written in Fortran code, which converts an EPN file into an XML file is described. Special consideration has been taken to convert all EPN variables to a meaningful XML equivalent. The program converts all the Versions of the EPN files. Together with the Fortran program the relevant Document Type Definition file (DTD file) used for the creation of the XML file is described.

Pulsar.dtd

The general definition of the converted structured data is defined in the Document Type Definition (DTD). It is obvious that special care has been taken so that the tags have a reasonable meaning for the untrained user. The DTD file is shown below:

```
<!-- pulsar.dtd                                                    -->
<!-- pulsar is the root of the document                            -->
<!ELEMENT pulsar (line1,line2,line3,line4,line5,
                  subheader1+,subheader2+,datas+)>
<!ATTLIST pulsar type (default|preserve) "preserve">

<!-- line1 contains information about the first line of the epn -->
<!ELEMENT line1 (version,counter,history)>

<!--version,counter,history are all elements inside line1       -->

<!ELEMENT version (#PCDATA)>
<!ELEMENT counter (#PCDATA)>
<!ELEMENT history (#PCDATA)>

<!-- line2 contains information about the second line of the epn -->
<!ELEMENT line2 (jname,cname,pbar,dm,rm,catref,bibref)>
```

```
<!ELEMENT jname (#PCDATA)>
<!ELEMENT cname (#PCDATA)>
<!ELEMENT pbar (#PCDATA)>
<!ELEMENT dm (#PCDATA)>
<!ELEMENT rm (#PCDATA)>
<!ELEMENT catref (#PCDATA)>
<!ELEMENT bibref (#PCDATA)>

<!-- line3 contains information about the third line of the epn  -->
<!ELEMENT line3 (rah,ram,ras,ded,dem,des,telname,
             epoch,opos,paflag,timflag)>

<!ELEMENT rah (#PCDATA)>
<!ELEMENT ram (#PCDATA)>
<!ELEMENT ras (#PCDATA)>
<!ELEMENT ded (#PCDATA)>
<!ELEMENT des (#PCDATA)>
<!ELEMENT telname (#PCDATA)>
<!ELEMENT epoch (#PCDATA)>
<!ELEMENT opos (#PCDATA)>
<!ELEMENT paflag (#PCDATA)>
<!ELEMENT timflag (#PCDATA)>

<!-- line4 contains information about the fourth line of the epn -->
<!ELEMENT line4 (xtel,ytel,ztel)>

<!ELEMENT xtel (#PCDATA)>
<!ELEMENT ytel (#PCDATA)>
<!ELEMENT ztel (#PCDATA)>

<!--line5 contains information about the fifth line of the epn   -->
<!ELEMENT line5 (cdy,cdm,cdd,scanno,subscan,npol,nfreq,nbin,
             tbin,nint,ncal,lcal,tres,fluxflag)>

<!ELEMENT cdy (#PCDATA)>
<!ELEMENT cdm (#PCDATA)>
<!ELEMENT cdd (#PCDATA)>
<!ELEMENT scanno (#PCDATA)>
<!ELEMENT subscan (#PCDATA)>
<!ELEMENT npol (#PCDATA)>
<!ELEMENT nfreq (#PCDATA)>
<!ELEMENT nbin (#PCDATA)>
<!ELEMENT tbin (#PCDATA)>
<!ELEMENT nint (#PCDATA)>
<!ELEMENT ncal (#PCDATA)>
<!ELEMENT lcal (#PCDATA)>
<!ELEMENT tres (#PCDATA)>
<!ELEMENT fluxflag (#PCDATA)>

<!-- subheader1 contains information about the subheader1
     of the epn                                                -->

<!ELEMENT subheader1 (idfield,nband,navg,f0,f0u,
                  df,dfu,tstart)>

<!ELEMENT idfield (#PCDATA)>
<!ELEMENT nband (#PCDATA)>
```

```
<!ELEMENT navg (#PCDATA)>
<!ELEMENT f0 (#PCDATA)>
<!ELEMENT f0u (#PCDATA)>
<!ELEMENT df (#PCDATA)>
<!ELEMENT dfu (#PCDATA)>
<!ELEMENT tstart (#PCDATA)>


<!-- subheader2 contains information about the subheader2
     of the epn                                                    -->

<!ELEMENT subheader2 (scale,offset,rms,papp)>

<!ELEMENT scale (#PCDATA)>
<!ELEMENT offset (#PCDATA)>
<!ELEMENT rms (#PCDATA)>
<!ELEMENT papp (#PCDATA)>

<!-- the data elements contains the data of the epn              -->
<!ELEMENT datas (#PCDATA)>
<!ELEMENT data (#PCDATA)>
```

Pulsar.xml

Using the DTD file mentioned above and the EPN data for PSR 1929+10 mentioned in page 24, an XML file has been created. This file is shown below. It contains structural units (elements like ‘line1, line2’, etc). Each line consists of header definition tagged data (‘version, counter’, etc), subheader definition tagged data (‘idfield, nband’, etc) and finally the sequence of data.

```
<?xml version="1.0"?>
<!DOCTYPE pulsar SYSTEM "pulsar.dtd">
<pulsar type="preserve">
<line1>
    <version>EPN 6.00</version>
    <counter> 222</counter>
    <history>     Data created by epos2epn v6.0
</history>
  </line1>
  <line2>
    <jname>1932+1059   </jname>
    <cname>1929+10     </cname>
    <pbar>    .22651808350</pbar>
    <dm>   3.176</dm>
    <rm>    -6.100</rm>
    <catref>tmlc95</catref>
    <bibref>studen19</bibref>
  </line2>
  <line3>
    <rah>19</rah>
    <ram>32</ram>
```

```
    <ras>13.900</ras>
    <ded> 10</ded>
    <dem>59</dem>
    <des>31.994</des>
    <telname>EFFELSBG</telname>
    <epoch> 51013.000</epoch>
    <opos>    .000</opos>
    <paflag> </paflag>
    <timflag>A</timflag>
  </line3>
  <line4>
    <xtel>    4033949.50000</xtel>
    <ytel>     486989.40000</ytel>
    <ztel>    4900430.80000</ztel>
  </line4>
  <line5>
    <cdy>2000</cdy>
    <cdm> 5</cdm>
    <cdd>11</cdd>
    <scanno>2121</scanno>
    <subscan>   1</subscan>
    <npol> 4</npol>
    <nfreq>   1</nfreq>
    <nbin>1024</nbin>
    <tbin>  221.000000</tbin>
    <nint>    66</nint>
    <ncal>   1</ncal>
    <lcal>  50</lcal>
    <tres>     .000000</tres>
    <fluxflag> </fluxflag>
  </line5>
  <subheader1>
    <idfield>LHC                              </idfield>
    <nband>   1</nband>
    <navg>   1</navg>
    <f0>     .000000</f0>
    <f0u> GHz    </f0u>
    <df>   40.000000</df>
    <dfu> MHz    </dfu>
    <tstart>14578478479.38538</tstart>
  </subheader1>
  <subheader2>
    <scale> .100000E+01</scale>
    <offset> .441750E+05</offset>
    <rms> .300701E+02</rms>
    <papp>   .226518094540</papp>
  </subheader2>
  <datasets>
    <data>
31B595F3F1A2E8CCFA32E8CCE9A0E9A0F115EEDFFD82EDC5F088EF26F728F3D8FFFFEDC5
ECAAECF0
ECF0E7F8F276DB8BDCECEB02EFFADD33F4F3E2FFF4ACF6E2ECAAF1A2F465ED37F4F3EFB3
EEDFF5C7
E9A0F465EFB3ECAAF580EE99F276F654E4A8EFFADD7A6A981F7B1B5622CB12390E1509F1
19AE0C26
08D6109116A513541E600E1501611617139B146F26A914B62C7628DE1A3C0BDF26A90F30
1C7128DE
```

0B99123929F91A3C223E23120F301B9D1BE417BF19F51C7112C7116514B610D817320B52
23121968
1CFF223E1C7120961732058506A017BF2F7F0EE9042423120E5C14FC161718931C710CFA
1E601921
130E072E1B10242D14B6184D20960A7E1B101B56104B17BF26A9072E091C09F11A82184D
0F761E60
184D1C2A1FC20D411EED184D16EB235912391BE41F7B1F342008154314B616A519AE08D6
109121F7
12C70A3721B020DC173210041617088F212312801F7B180623E617321AC90EE9280A239F
12C71921
154311F31FC215430EE92E1E26A91BE4142822CB1D45142809F11D4513E20A7E0D41184D
0D8817BF
204F239F165E0BDF0963165E18060AC51CFF111F0FBD2A4010041E19111F161711652123
1F3415D0
142826F019AE21B015D01F34139B130E1BE40FBD14281CFF0E5C1893184D1AC918931EED
216A216A
2359216A0EE90B991DD3154318DA1FC214FC21B01D8C14281EA71CFF15D01F341AC90DCE
0D41300D
21230D8812390C261BE41A8217BF184D22842736139B2898130E1004104B1C2A1F7B0E15
2B5B1806
158A19F51E191B100FBD000013E21B56135412391D4511651EED011A1C2A1091111F1428
0DCE184D
1B9D18DA10912C76116517BF1BE41E6029B320DC146F1B1020961BE417791CFF21232096
10911806
0E5C19F51BE41C711A821BE403971A821B101B9D20DC1F3423E61C710F761AC9192113E2
11F31354
1C2A1B101B9D10910EE9239F184D1B56111F1C2A19AE15430E5C1E191F3419AE16A514B6
0C6D2096
242D1B561E601E19189319F50EE908D6088F16A5204F146F196810D81C2A15431E19204F
11651239
111F0EA21732189300D415D01B9D18DA25D506A0242D1C71177912C719F51A3C17321239
16171CB8
139B146F21F71CFF165E1E1912C719F514FC111F20960B5214FC1D8C12C71E1915431F7B
2CBC0BDF
1F7B1B100E5C1FC21D8C250118931428280A223E184D100417791EED084806A01F7B165E
1CB82123
111F1D4511F3091C139B1B9D00461B9D1D8C2473184D0EE91B5617BF0E5C1B5613541D45
1DD318DA
1893184D109106A009AA1B9D146F223E19AE0EE91C7111650EA229B30D8826A917BF14FC
1EED1239
19210CB4189309AA1D4510041091173221F71EA70CFA21230E15111F111F29B30FBD2ACD
17791B10
0F3007BB20DC12391FC21AC917BF1B56146F0AC521B029B31893239F20DC142819212925
146F2B14
2D03189316EB196816171E600FBD196818930B0B1C2A12C715D0158A1806142808480DCE
0D8835D9
13E214FC1AC91B5620DC09F117790EA20A371CB825D519F51A3C1B102736258E142813E2
20081893
1A8223E6072E17BF14FC1EED1280239F1C2A24BA09F11CB8189314FC24BA10041A3C2DD7
1D451AC9
19F5165E1165223E29251EA728511D451893139B16A51CB8223E1AC90C6D273619681E60
0EA207BB
1D451D8C242D29251A3C1F7B19F5046B16EB228426F02312111F1091184D2096239F1BE4
2A870EE9
223E21B01F7B189320DC2096261C14B60D412D0311F321F70F3026F02CBC1428158A2501
24BA2925

17321B1012C71B561A3C1B9D19F51EED25012C2F296C266226A93D95385545973CC14058
47CC34BE
58A563235ADA59C052D851BD57D15D575BAF617B61C2697D73B5720D851B66BA6EBD50A3
5B684F88
3F3D5C3C44C347CC39FE316E343133EA35D9362025D52F38389C2501335D2E1E20963781
13541617
242D26A914B630E120081CB8173221B0231218931EED2CBC1AC91E60130E18DA06E706E7
0DCE21B0
1BE41AC90D88123911AC184D1C2A29251F7B1D4523E609AA09630B5215431FC21FC21239
1D451F7B
10D811AC1C711E191B56135423121617158A22840CB4072E109109AA16A512C71968204F
15430963
14280E150E5C1D8C10D80EE9192120DC0CFA139B1732223E11AC31B5192122841A3C1091
17BF0B52
1AC91B9D23E618DA173218DA228410D82BA1139B084814B6139B0E151543277D14B61D8C
1CFF2736
16171B9D1B5624730D880C261C711EED14B617BF2C2F0DCE22CB21F71B5611F31EA71E60
139B1A82
1617135415D016A50C6D123909F124731A3C200813541CFF13E2239F1239109128981AC9
20DC14FC
130E1BE417BF13E21F7B2ACD1B5620962359158A158A239F1806139B18060EE916172008
1004146F
11F319211F3423591F7B16EB13E229251EA71F3419AE0D88204F192112391E19111F19AE
0EA21921
1D8C0EE919210CFA24BA0E5C130E1C7113E220DC1A3C1C2A200820DC14FC1732000016EB
03DD15D0
1B560FBD1CFF216A07741280277D15D0161714B623591D8C1EA71A3C23121DD30DCE1A3C
0DCE1E19
2D031F7B19211D451F3421F7165E11651921216A16A504F81DD310D818931968196819F5
1B9D0CFA
184D200814FC21F711F314FC11F30E150FBD0B0B12801B9D07740EA2158A0B52165E0B52
184D1E19
139B0EA21C2A173216A52B140D4118DA1617123910D8111F139B0E5C1CB8146F0A370963
11AC1C71
15430F301B9D1FC20EE91004200805CC104B18DA19AE0C261E1920DC0E5C18DA11F31779
16A5091C
06A0216A2925123914B615431FC2231222840EA2261C19AE165E209613E212801C2A1D8C
1AC9158A
1B9D0D88128001A800000000000000000000000000000000000000000000000000000000
00000000
  </data>
  </datasets>

Bellow is shown a part of the XML code of the preceding example rendered by MS Explorer 5.5.

Access to the program

The program files and the libraries, the DTD file, an EPN-file and its XML equivalent can be found in http://www.astro.auth.gr/~jhs/XML. A large collection of EPN data can be found in: http://www.mpifr-bonn.mpg.de/div/pulsar/data.

XML APPLICATIONS

Several tools, developed for XML tagged data, have already been created. Among them:

- The MS Explorer (V>"5.5) already reads (by simply clicking on the xml-file) XML formatted data. Other browsers are expected to follow.
- An international agreement on an XML standard (VOTable) for exchanging astronomical tables, including FITS format, has been met (Genova, 2002).
- The Cambridge University Press is developing publication tools based on emerging technologies such as XML (Business Weekly\2001).

CONCLUSION

In the beginning of this Diploma Thesis an extensive introduction to the physical parameters of pulsars is presented, followed by a description of the EPN format. It is also described an integrated procedure for converting EPN data to XML format. The Fortran code and the DTD file needed for the conversion is adequately described. It is obvious that special care has been taken so that the tags have a reasonable meaning for the untrained user. It is also important to mention that the hexadecimal format of the EPN data has been retained. A few applications using XML formatted files have been identified and presented in §6. It is anticipated that in the future XML formatted files will play a distinct role in the fast transfer of readable data between internet nodes and also in the efficient response to search engine capabilities.

Acknowledgements

I would like to thank:

Professor J.H. Seiradakis, my supervisor, who guided me through this Thesis.
My colleague, Marios Chatzikos for his consulting help on programming and for his collaboration during the presentation of my diploma thesis.
Alexis Karatzoglou for giving me the first useful hints at the beginning of the project.
My parents for supporting me all these years.

Bibliography

1. Σπύρου, N. K. 1995. *Αρχές Αστρικής Εξέλιξης*. Τμήμα Φυσικής, Πανεπιστήμιο Θεσσαλονικής.
2. Σειραδάκης, I.X. 1996. *Σημειώσεις Ραδιοαστρονομίας*. Τμήμα Φυσικής, Πανεπιστήμιο Θεσσαλονικής.
3. Manchester, N and Taylor, H. *Pulsars*. San Francisco, CA: W. H. Freeman, 1977.
4. Lyne, A. G. and Graham-Smith. *Pulsar Astronomy*. Cambridge, England: Cambridge University Press, 1990.
5. Lorimer D.R., Jessner A., Seiradakis J.H.,et al., 1998,A&AS 128,541
6. Genova F., 2000, On line information in Astronomy in Creating a European Forum on Open Archive
7. Business weekly, 2001, http://www.businessweekly.co.uk/news/view_article.asp?article_id=5040"
8. www.w3.org/TR/2000/REC-xml-20001006
9. xml.gsfc.nasa.gov/XDF/XDF_home.html
10. www.ibm.com/developerWorks
11. www.edpsciences.com/articles/astro/full/1998/06/h0535/node1.html