The NISP tools

Users Manual

Jens Stavnstrup

The NISP tools: Users Manual

Jens Stavnstrup

Published 2009-12-06 21:26:12

List of Figures

Preface

This document describes the tools necessary to create HTML and PDF versions of the using sources of the documents implemented using the DocBook Document Type Definition (DTD). The following chapters describes how to install install and also cover a brief description of the most common DocBook elements used by the documents. For a more comprehensive understanding of the technologies used a bibography is included.

Part I. Quick Guide

1. Installation and Configuration

Working with DocBook XML files, requires quite a bit of software. However, the number of packages necessary, depends on the requirements of the user. To simplify things, most of the tools have been bundled in the nisp-tools package. In order e.g. to generate a HTML version of the , the users only have to install Java, the nisp tools distribution and the sources for the XML version of the . A detailed description of all the different tools included in this package can be found in Appendix A, *Inside the NISP tools distribution*.

All software packages can in principle be installed anywhere, but everything should work without a glitch, if some of the packages are installed in a couple of predefined places.

File paths in archives

The user should be aware, that some distributions are archived inside a sub directory, and some are not., e.g. the NISP tools distribution is unpacked into the sub directory nisp-tools-3.1, where all the files in the NISP tools distribution are located. It is up to the user to ensure this issue is resolved properly. So if the installation instructions recommends that the NISP package should be installed under Windows in the directory `c:\nisp-tools`, the path to the `README` file from the NISP tools distribution, should be the file path `c:\nisp-tools\README` and not the file path `c:\nisp-tools\nc3ta-tools-3.1\README`.

1.1. Installation of Java Runtime Environment 1.4.2_06

The installation of the Java 1.4.2_06 is straight forward. The Java 2 Software Development Kit or the Java 2 Run-time environment comes as an archived executable, and is by default installed under windows at `c:\jdk1.4.2_06`. If the user prefers to use another version of java, please consult the installation instruction of the programs Xerces, Saxon, Batik, Fop, Sun catalog resolver and Ant to identify the system requirements for these tools.[1]

The user *must* always add the directory, where Java executables are located to the `PATH` environment variable. Typical values for `PATH` are:

Java 2 Software Development Kit

[1]Note, that the tools have not been tested with Java 2 SDK 1.4, so you are entirely on your own, if you choose this platform.

```
c:\jdk1.4.2_06\bin            (Win32)
/usr/java/jdk1.4.2_06/bin     (Unix)
```

Java 2 Runtime Environment

```
c:\Program Files\JavaSoft\JRE\1.4.2_06\bin     (Win32)
/usr/java/jdk1.4.2_06/bin                      (Unix)
```

The user should also set the environment variable `JAVA_HOME`. Although normally not required by Java 1.4.2_06, the Ant program requires this variable set for reasons of backward compatibility. Typical values for `JAVA_HOME` are:

Java 2 Software Development Kit

```
c:\jdk1.4.2_06            (Win32)
/usr/java/jdk1.4.2_06     (Unix)
```

Java 2 Runtime Environment

```
c:\Program Files\JavaSoft\JRE\1.4.2_06     (Win32)
/usr/java/jdk1.4.2_06                      (Unix)
```

1.2. Installation of NISP Tools

The NISP tools distribution consists of all the packages necessary to build a HTML version of the . The NISP tools can in principle be installed anywhere, but if the user are runs a *Win 95*, *Win 98* or a *Windows ME* platform, then the path to the distribution should be relatively short due to a bug in the command processor.

The user should set the environment variable `NISP_HOME` to the directory, where the NISP tools package, was installed. Typical values for `NISP_HOME` are:

```
c:\nisp-tools-3.1            (Win32)
```

`~/nisp-tools-3.1/` (Unix)

Using a generic root name

Since the NISP tool package may be released frequently, the user could rename the tools directory to e.g. `c:\nisp-tools`, and of course set the `NISP_HOME` environment variable accordingly.

1.3. Installation of the NISP sources

The NISP XML sources (`nisp-src-3.0.zip`) *must* be unpacked in the empty `src` directory in the tools distribution.

1.4. Installation of the HTMLHELP compiler

The current webversion of the NISP also comes with a HTML Help version. This requires installation of the Microsofts HTML Help compiler, which due to license restrictions can not be included in the NISP tools package. The user is required to download this distribution from the Microsofts website. Search for "html help downloads".

The configuration file "user.properties", which is located in the root of the tools distribution contains the property "hcc", which must point to the location of the HTML Help Compiler.

When you use the default installation using the file "htmlhelp.exe" distributed by Microsoft, the HTML Help Workshop will be installed in "C:\Program Files\HTML Help Workshop". The HTML Help Workshop also includes the HTML Help Compiler.The default setting of the property "hcc" in the file "user.properties" therefore is "hcc = C:\\Program Files\\HTML Help Workshop\\hhc.exe".

Change the value of this parameter, if your version of the HTML Help Compiler is located in a different directory.

Please note that the file "hcc.exe" itself is not sufficient to be able to compile a HTML Help project. At least the following three files, which are included in the Microsoft distribution of the HTML Help Workshop, are required to be able to compile a HTML Help project:

- hhc.exe. Default this file is installed in C:\Program Files\HTML Help Workshop.

- itcc.dll. Default this file is installed in C:\Program Files\HTML Help Workshop.
- hha.dll. Default this file is installed in C:\Windows\System32 (on WindowsXP).

If you do not have sufficient privileges to install the HTML Help Workshop, you can copy these files to the same folder, e.g. D:\Tools\NISP\HTMLHelp and change the "hcc" property in the file "user.properties" to e.g. "hcc=D:\\Tools\\NISP\\HTMLHelp". Now it is possible to invoke the HTML Help compiler and generate the ".chm"-file.

This approach has one problem: the index will not be generated correctly. You must register the file "itcc.dll" to be able to generate a correct index as well. If you have sufficient privileges, you can perform this action using the command "regsvr32 itcc.dll".

1.5. The XSL/XSLT stylesheets distribution

With the release of the tools distribution `nisp-tools-3.1.zip`, and additional file called `nisp-xsl-3.1.0.zip` was also released. This file contains the stylesheets used by the tools distribution, and is by default already included in the tools distribution.

Whenever an update to the stylesheets is released, the revision number of the stylesheet will be increased. So the first new version would e.g. be named `nisp-xsl-3.1.1.zip`.

1.5.1. Updating the stylesheets

To upgrade the stylesheets of the tools distributions, the user should just delete all files in the tools directory `xsl`, and unpack the new stylesheets into that directory. Since the original stylesheets from the tools distribution `nisp-tools-3.1.zip`, was also distributed in the file `nisp-xsl-3.1.0.zip`, the user can always return to the original sets of stylesheets.

Part II. Advanced

2. The standard database DTD

The current NISP standards database is implemented as a XML document. The original design of the database reflected the structure of the list of standard and profiles described in the NC3 Technical Architecture version 1. In order to enable consistency across the different volumes, the database have been extented to be able to describe the selection of mandatory-, emerging- and fading standards and profiles. These selection are justified in the traceability matrix included in the rationale document, which are also part of the database. Recently the database have been reorganized to reflect additional requirements, but also to prepare the database for inclusion in the NATO Architecture Repository (NAR)

In order to enable validation of the database a schema in form of a DTD have been defined. This DTD is located in the source distribution at `schema/dtd/stddb40.dtd`. Since the database contains DocBook fragments such as paragraphs, unnumbered lists etc., the standard database DTD have been implemented as an extension to the DocBook DTD. Ideally we should use multiple namespace in order to avoid name conflicts, but that is very complicated when using a simple schema language such as DTD. A better approach will be to implement the standard database schema in a modern schema language such as Relax NG.

In the following pages, the syntax of the database is described in detail and when appropiate, examples of the actual implementation is included.

The standard database DTD is logically separated in three different parts:

- standards taxonomy
- standard and profile description
- standard selction

It could be tempting also to separate the database into three physical parts - although not necessary in three different files. However

2.1. Standard taxonomy

The standard taxonomy describes how the standards and profiles are organised and is based on the taxanomy decribed in the NNEC Feasibility study

The root element of the DTD is the element standards.

```
<!ELEMENT standards (servicearea+, externalrefs?, organisations)>
```

The standard database represented by the standards element consists on one or more servicearea and some eksta elements used for different purposes.

```
<!ELEMENT servicearea ((servicecategory |  standardrecord |
                        profilerecord | referencerecord | sp-list)*)

<!ATTLIST servicearea
          id ID #REQUIRED
          mid CDATA #IMPLIED
          title CDATA #REQUIRED>
```

A servicearea[1] may contain 0 or more servicecategory, standardrecord, profilerecord, referencerecord or sp-list elements. As servicearea element must have an id and title attribute.

```
<!ELEMENT servicecategory ((category | standardrecord | profilerecor

<!ATTLIST servicecategory
          id ID #REQUIRED
          mid CDATA #IMPLIED
          title CDATA #REQUIRED>
```

A servicecategory[2] element may contain 0 or more category, standardrecord, profilerecord, referencerecord or sp-list elements. As servicecategory element must have an id and title attribute.

```
<!ELEMENT category ((standardrecord | profilerecord | referencerecor

<!ATTLIST category
          id ID #REQUIRED
          mid CDATA #IMPLIED
          title CDATA #REQUIRED>
```

[1]A servicearea element should properly only contain servicecategory elements, but the initial database have all these elements.
[2]A servicecategory element should properly only contain category elements, but the initial database have all these elements.

A category element may contain 0 or more standardrecord, profilerecord, referencerecord or sp-list elements. As service category element must have an id and title attribute.

2.2. Standard and profile definition

Both de-jure and de-facto standards from many different standard *organisations* are labeled different. Some standards have multiple parts each have a standard number, where for other standards only the cover standard have a number. Some standards are registered by multiple standard bodies. Some standards are updated, but the actual update are released as a separate document instead of releasing a new version of the stabdard. The same is true for profiles.

To distinguish between information about a standard provided a standard body and infomation only relevant to this database, an element standardrecord is used to describe a standard and also to describe historical aspect of the inclusion of the standard in the database. A profilerecord element is similar introduced to describe a profile.

In the NISP volume 3, a referencerecord element, was used to refer to a standard or profile, which was described in another location in standard hieracy.

```
<!ELEMENT standardrecord (debug?, ((standard | applicability | statu

<!ATTLIST standardrecord
          tag CDATA #REQUIRED
          id ID #IMPLIED>
```

A standardrecord element may contain a debug[3] element followed by 0 or more standard, applicability or status elements[4]

A standardrecord element must contain an tag attribute and may contain an id attribute.

```
<!ELEMENT profilerecord (debug?, ((profilenote | parts | applicabili
                          sremark))>

<!ATTLIST profilerecord
          tag CDATA #REQUIRED
          id ID #IMPLIED>
```

[3]The historical reason of this element is unknown, maybee a stylesheet will reviel the reason.
[4]This looks strange

A profilerecord element may contain a debug element followed by 0 or more profilenote, parts, applicability or status elements. Or a profilerecord may contain a debug element followed by an sremark element.

A profilerecord element must contain an tag attribute and may contain an id attribute.

```
<!ELEMENT referencerecord (sremark)>

<!ATTLIST referencerecord
          tag CDATA #REQUIRED>
```

A referencerecord element contains an sremark element. A referencerecord must contain a tag attribute,

```
<!ELEMENT standard ((correction | alsoknown | comment)*,  parts?)>

<!ATTLIST standard
          orgid  CDATA #REQUIRED
          pubnum CDATA #REQUIRED
          date   CDATA #REQUIRED
          title  CDATA #REQUIRED>
```

A standard element may contain 0 or more correction, alsoknown or comments elements possibly followed by a parts element.

A standard element must contain the attributes orgid, pubnum, date and title.

```
<!ELEMENT correction (#PCDATA)>

<!ATTLIST correction
          cpubnum  CDATA #REQUIRED
          date     CDATA #REQUIRED>
```

A correction element

```
<!ELEMENT alsoknown (#PCDATA)>

<!ATTLIST alsoknown
          orgid   CDATA #REQUIRED
          pubnum  CDATA #REQUIRED
          date    CDATA #REQUIRED>
```

A alsoknown element contains a describtion which might be empty. A alsoknown element must also contain a orgid, pubnum and date attribute.

NATO stanag 3809 for Digital Terrain Elevation Data (DTED) is also know as ISO/ IEC 8211.

```
<!ELEMENT comment (#PCDATA | ulink)*>
```

A comment element contain data in the form of text or DocBook ulink elements.

```
<!ELEMENT parts (standard)+>
```

The parts element contains one or more standard elements.

```
<!ELEMENT profilenote (#PCDATA)>
```

The profilenote element contains text.

```
<!ELEMENT applicability %ho; (%tbl.entry.mdl;)*>
```

The applicability element uses the same content model as the DocBook element entry.

```
<!ELEMENT status (info?, uri?, history)>
```

```
<!ATTLIST status
          mode   (unknown|rejected|retired) #IMPLIED "unknown"
          stage  CDATA #REQUIRED>
```

The status element contains in the following order an info, uri and history element. The history element is mandatoty, the infor and uri elements are not. The uri element is defined in the DocBook DTD.

The status element contains the attributes mode and stage. The first is implied and the second is required.

```
  <!ELEMENT info (#PCDATA | ulink)*>
```

The info element contains textual information, which is not appropiate to put in e.g. the applicability element. It can also contain ulink elements, whiche e.g. could point to the standard.

```
<!ELEMENT history (event)+>

<!ELEMENT event (#PCDATA)>

<!ATTLIST event
          flag    (added|changed|deleted)   #REQUIRED
          date    CDATA   #REQUIRED
          rfcp    CDATA   #IMPLIED
          version CDATA   #REQUIRED
>
```

The histoty element contains historical infomation about a standard. When was it added, changed or deleted and through which RFCP. The history element contains of one or more event elements.

The event element may contain a rfcp attribute and must contain the attributes flag, date and version.

The date attribute must use the following wellknown pattern YYYY-MM-DD.

2.3. Standard Selection

Part III. Old manual

3. Introduction

The NISP tools is a collection of software packages, XSLT stylesheets etc., used to build a HTML- and a version suitable for printing of the NATO C3 . The XML source files are not included in this distribution, but are distributed separately.

3.1. Background

During the 16th NOSWG meeting in Feb 2001, NC3A informed the group, that in the future the Microsoft Word version of the would no longer be maintained, but instead all future editing of the architecture documents would be done on the HTML (Hyper-text Markup Language) version of the documents. Although it is nice to have a HTML version of the architecture, it often just as convenient to have a printed version of a properly formatted document. However, since the burden of maintaining two versions in parallel requires a substantial amount of resources, I proposed a different solution, such as using a format that focus on the contents of the document, as e.g. using a structured markup language instead. The idea being, that our limited resources are better utilised focusing on the content rather than the presentation of the content. In order to prove the concept, I began to investigate the state of the art in this field. This paper describes the result of this work.

3.2. SGML & XML

The obvious solution was to use something like the SGML (Standard Generalised Markup Language). SGML is a meta-language, which describes how markup languages are defined. Unlike HTML, SGML does not consists of any predefined elements or semantic rules, but are strictly used to define markup languages of which HTML is the most popular. But where HTML also contains tags, which describes font-size, color etc. Meta-Markup languages consists of elements, that only are used to the describe the logical structure of a document. In the documentation domain these elements could be elements, like chapter, sections, paragraphs, unnumbered lists etc.

One solution, could be to look elsewhere in NATO. During the 2nd meeting in the XML Ad-Hoc Group, NAMSA presented a solution to the problem of managing huge amounts of documentation by using a homegrown document format. Although I could have chosen a similar approach, I decided, that the proper way would be to use a format that already enjoys considerably support from miscellaneous publishing houses.

One popular SGML implementation is the markup language DocBook, which is very well suited for technical documentation. There is one strong argument against adopting the markup language to mark-up the . Although SGML has been around for quite a while, and is very well suited for handling huge repositories of technical documentation, the tools for editing and validating SGML documents are very complicated and therefore also very expensive. However, just a couple of weeks before the NOSWG

meeting, OASIS released a XML version of DocBook. Since XML is much less complicated than SGML, it is very likely, that inexpensive tools for editing XML documents will be available to the public, within a relative short time frame.

3.3. Parser's and other tools

Since XML is a moving target, the selection of tools can be quite a complicated affair. Releases are frequent, and new and not so mature tools arrive every day. Fortunately, most of the tools necessary for the translation of the , are quite stable, and require almost none if any post-processing by the user. When working with a platform-independent technology such as XML, it can simplify matter a lot, if the tools used would also be platform independent. This can be accomplished by selecting tools written in Java, which at least in theory is platform-independent.

Since the major task in processing the NC3 documents are the transformation of the XML documents to HTML and a format suitable for printing, we need a tool that implements the XSLT 1.0 standard. Among the most popular tools are Xalan from the Apache Group and Michael Kay's Saxon. Both of these tools are implemented in Java, and Saxon is currently considered more mature, and also implements most of the additional functionality defined in the working draft XSLT 1.1, of which we need the ability to merge multiple documents, a property not defined in the original XSLT 1.0 specification.

Before the transformation of an XML document is invoked, the document must be parsed by an XML parser. Saxon comes preinstalled with the AElfred parser, which is not a validating parser, required in order to ensure that all NISP documents conform to the DocBook specification. Most of the original XML parsers, are not validating parsers, but recently the Apache Group have released the validating parser Xerces.

Currently there is no open source implementation, which does a really good job in creating a printed version of the XML documents. The best open-source project is the FO processor *Fop*, which currently is undergoing a major redesign effort and the frequency of releases have therefore dramatically slowed down. There are a couple of very good commercial implementations, which suffer seriously from the problem, that they are very expensive.

Although there are many bugs and facilities currently unimplemented in the FO processor. It turns out, that most of the problems, which are relevant to the documents can be temporary solved by manipulation of the FO tree, before the layout and rendering phase resulting in a PDF document.

3.4. Contents of this document

The next few chapters describes, how to install the necessary tool, and discusses XML, XSL, DocBook and the different tools in the distribution.

4. Introduction to XML

XML is a subset of SGML, and are designed for the Web. Like its ancestor SGML, XML do not have any predefined tags, but leave it up to the users to define their own vocabulary. The XML standard as defined in [XML 1.0], describes a set of rules that must be followed when writing XML documents. The following sections will very briefly describe the structure, and the usual components of a XML document. Additional information can also be found in the books [HaMe01] and [Ma01]. The user can also read the articles [Wa98] and [Wa01b] online.

4.1. The building blocks of XML

A XML document consists of a number of syntactical constructs, which will be defined in the following.

4.1.1. XML Prologue Statement

All documents may and should begin with a XML prologue statement

```
<?xml version="1.0"?>
```

This statement must be placed on the first line of any XML document and might besides the version attribute, contain additional attributes such as the encoding- and standalone attribute.

4.1.2. Elements

The main building block of XML documents is the element. An element is delimited by a *start tag* and an *end tag*. Everything between the start tag and the end tag is the *content* of the element. Every XML document is a hierarchical structure with exactly one root element and a number of children elements. In the example document below taken from [HaMe01], the root element is `person` delimited by the start tag `<person>` and end tag `</person>`.

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>computer scientist</profession>
  <profession>matematician</profession>
```

```
  <profession>cryptographer</profession>
</person>
```

The hierarchical structure of XML is obvious, if we think of the person element as the *parent element* containing four *children elements*, i.e. one `name` element and three `profession` elements. The `name` element again consists of two *children elements* `first_name` and `last_name` element, and each of the `profession` elements consists of only *character data*. XML tags are case-sensitive, i.e. `<person>` and `<PERSON>` are two different start tags.

When the example above is parsed by a XML parser, besides the elements and the prologue statement, the parser will also identify some *white space*, e.g. the spaces before the start-tag `<first_name>` and also after the end-tag `</first_name>`. This means that there is actually more that four children elements, contrary to what was stated before. Whitespace can be ignored by the XML parser, but sometimes it can be impossible for the parser to determine whether some space are semantically important or not, unless a document is also *validated*. In the DocBook DTD, this can be significant, when we are working with table cells.

If an element does not contain other elements but only attributes, then it can be written using the shortcut <myelement/>.

4.1.3. Attributes

Each element can contain one or more attributes in the form of a key value pair. Keys and values are separated by an equals sign and values are enclosed in single- or double quotes.

```
<person born="1912/06/23">
  Allan Turing
</person>
```

4.1.4. Processing Instructions

It is sometimes necessary to provide an application which processes a XML document with information, which for practical reasons is not part of the DTD (see Section 4.3, “DTDs”) for that document. For this purpose an author can use a *processing instruction* (PI) as an alternative means to provide information to the processing application.

A processing instruction begins with <? and ends with ?>. Immediately following the <? is an XML name called the *target*, identifying the application for which this PI is used, the target can also be considered the name of a PI. All the text to the right of the target is in a format appropriate for the application the PI is intended for.

In volume 3, 4 and the rationale document of the , PIs is used to identify a collection of standards, which will be merged into the final document. One example of such a PI used in volume 3 is shown below.

```
<?dbmerge cid="se-lng"?>
```

Whenever the stylesheet `xsl/merge3.xml` meets such a processing instruction, the appropriate standard is selected from the standard database `standards/ta-standards.xml` located in the s source distribution.

4.1.5. Comments

Comments in XML are syntactical identical to the comments in HTML. Comment are used only to comment the document. A XML parser is not required to pass the comments up to the application, so comments should only be used for informational purposes. Comments can appear anywhere in a document, except inside tags and inside other comments.

```
<-- This is an example of a comment -->
```

4.1.6. Entities

When parsing a XML document, it is sometimes necessary to use characters normally reserved for marking-up a XML document, such as the bracket characters ('<' and '>') which are used to embed tags. In order for the user to be able to use these two characters and a couple more, the XML standard have defined a concept called entities, which is a kind of macros, that are replaced with the appropriate characters, by the XML parser, when parsing a XML document.

This following list illustrates the list of predefined entities in XML

`<` The less-than sign, or opening angle bracket (<)

`&` The ampersand (&)

`>` The greater-than, or closing angle bracket (>)

`&quot;` The straight, double quotation marks (")

`&apos;` The apostrophe, or single quote (')

These five entities is central in the mark-up of XML documents, so whenever a user needs to write the character '&', this can be accomplished by writing the entity instead, e.g. '&'.

The DocBook DTD comes with a large number of predefined entities used to represents uni-code characters and other symbols, which can be used in DocBook documents. Besides the convenience of enabling access to a larger number of characters, normally not supported by text editors today, it is also sometimes safer although quite tedious to use these entities, instead of utilising the functionality of a given editor.

Entities comes in many flavours, and we will not go into detail, in this slightly complicated subject, but only mention the use of entities in the . For a more detailed description of entities, the reader are referred to the XML standard or the articles ??? and ???

- In most of the volumes, we do not referrer to the filenames of externally defined images, but instead uses parametric entities defined in the external file `figures.ent` (which is automatically is loaded, by the XML Parser to resolve the entities). The makes it much easier to keep track on the images used in the different volumes.

- In the source distribution, we have specifies the version number of the volumes in the file `VERSION`, which may be used by the different volumes, in the revision history.

- The same `VERSION` information is also used when building the distribution packages.

4.2. Structure of XML documents

A XML document contains one or more elements delimited by start- and end tags. There is exactly one root element, and all other elements are properly nested within this root element. Every attribute in an element must be embedded inside single or double quotes. These simple rules are central to the understanding of the structure of generic XML documents.

4.3. DTDs

A Document Type Definition (DTD) describes the legal structure of a document and is an integral part of the XML standard. The DTD describes the formal syntax of a given document and describes precisely, which elements and attributes are allowed, and the

context in which the elements are allowed. The DocBook DTD can make statements, such as a *sect2* element must be contained inside a *sect1* element, Or a *row* element are only allowed to contain one or more *entry* or *entrytbl* elements. It is not a requirement, that the DTD is available when writing XML documents, but whenever people are involved in the process, it is always a good idea, to use a DTD in order to ensure that the XML document is legal. The source distribution comes with a number of DTDs, which are described futher in Section 7.1.4, “DTD”.

4.4. Well-formed and Valid Documents

A XML document is well-formed if it follows the rules stated in Section 4.2, “Structure of XML documents”.

A XML document is valid, if it is well-formed and follows a syntax defined in a Document Type Definition (DTD). The description of an DTD is defined in the XML specification, and will not be described further in this paper.

4.5. Document Type Declaration

In order for an XML parser to validate a document written in a given DTD, a document type declaration must be included at the beginning of a document. The document type declaration is not required in a XML document, but when working with large DTD like DocBook, it is absolutely recommended.

A doctype declaration for a DocBook document could look like this:

```
<?xml version="1.0"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.5//EN"
     "http://www.oasis-open.org/docbook/xml/4.55/docbookx.dtd">
```

The declaration indicates, that the DTD used are specified by the PUBLIC identifier: `-//OASIS//DTD DocBook XML V4.5//EN`, the root element is book, and the SYSTEM identifier identifies the physical location of the DTD, which in this case is a web server at OASIS. The PUBLIC identifier are frequently used, when working with SGML documents, where through some magic, the PUBLIC identifier are resolved into the physical location of the DTD. The PUBLIC identifier is optional in XML documents, and the XML specification do not give any means, on how to resolve a public identifier, but this may change in a future version.

The SYSTEM identifier is mandatory for XML document and is always an URI. However, many parser's accept a file name, and then interpret this as an URI with the prefix `file://`

If the SYSTEM identifier is '`dtd/tongle.dtd`', this would indicate to the parser, that the documents DTD is called tongle.dtd, and is located in the sub directory dtd, which relative to the directory, where the document resides. With the introduction of the resolver library in the tools, it is now possible to use XML Catalogs[Wa02a] to resolve public identifiers.

4.6. Namespaces

Namespaces [Names] are primary used to distinguish between elements from different applications of XML, that share the same name. Namespaces are implemented by attaching a prefix to elements and attributes belonging to the namespace. Each of these prefixes are mapped to a URI, which uniquely defines the namespace. The URI may, but is not required to point to an actual document.

In the namespaces are used mostly in stylesheets, and are used to distinguish between stylesheet elements and elements used to generate the target documents, i.e. elements that belongs to the HTML and the XSL-FO namespace.

Although we do not define our own namespaces in the documents, we use multiple DTDs. It is therefore important to understand that a namespace and a DTD is in no way related. We can use DTD's without using namespaces and vice versa. Similarly we can work with documents, without using a DTD or namespaces. So although we have defined a DTD for standards used in the , this DTD, which is an extension to the DocBook DTD, do contains elements, which are part of the DocBook namespace and also elements which does not belongs to any namespace.

4.7. Character-Set in XML documents

All XML documents uses the Unicode character-set[UCS], which contains all the characters used today, and even characters from dead languages like Sanskrit and the Egyptian hieroglyphs. A character-set is actually a mapping from a character to a number[1], since computers only understand numbers in the form of bits and bytes. E.g. in Unicode the character 'A' is represented by the decimal number 65.

A *character encoding* describes how the characters of the text is represented, and also, which characters from the Unicode character set is included in the encoding. All the uses the ISO-8859-1 character encoding scheme, which according to [HaMe01] consists of "*ASCII character set plus accented letters and other characters needed for most Latin-alphabet Western European languages.*"

[1]Such a mapping is also called a *code point*

5. The DocBook DTD

The DocBook format [WaMu99] and [WaMu05] is an international SGML documentation standard currently managed by an OASIS technical committee. DocBook is actually a Document Type Definition (DTD), which is a syntactically description on how document compliant with the DocBook DTD should be written.

The first XML implementation of DocBook XML (4.1.2) was officially released by OASIS in February 2001, and is almost identical to the SGML version. There are of course some things, which currently cannot be expressed in XML, but this is not significant for the XML implementation of the .

5.1. Basic Markup Concepts

Using a DTD like DocBook will properly in the beginning be a frustrating experience comparing to using a GUI word processor like Microsoft's Word. The most important difference is that the usual control of layout and formatting is not possible. Markup is not about style, but strictly about contents. At first, this may seem like a major inconvenience, but in the long run, this is a actually a huge advantage, since it prevents adhoc definitions of styles, which might be changed later, and instead enables the use to concentrate on the contents of the document. The separation of contents from layout and style enable publication on many different medias, many not even considered by the time of the writing. It will also enable computer based manipulation of documents, which are currently only possible with the greatest difficulty using the current tools.

5.2. Used DocBook Elements

All-though the DocBook XML DTD consists of almost 400 elements. The XML implementation of the uses only a very limited subset of all the elements defined in the DTD as illustrated in the following table[1], which is based on Version 5.0 of the XML (dated December 15, 2003). The table clearly illustrates, that the elements mostly used, are actually related to table elements, and the paragraph element *para*.

The elements used in the is as described above only a small subset of all the elements in the DocBook DTD. In [WaMu05], a breakdown of the logical stucture of DocBook elements is described.

Element	**Vol. 1**	**Vol. 2**	**V2 Sup1**	**V2 Sup2**	**Vol. 3**	**Vol. 4**	**Vol. 5**	**Rationale**	**IHB**
Book Elements									
book	1	1	1	1	1	1	1	1	1
preface	0	0	0	0	1	1	1	0	1

[1]This table was generate using a perl program. See the file `extra/perl/estat.pl` in the tools distribution.

Element	Vol. 1	Vol. 2	V2 Sup1	V2 Sup2	Vol. 3	Vol. 4	Vol. 5	Ra-tionale	IHB
Navigation Elements									
index	0	0	0	0	1	1	0	0	0
indexterm	0	0	0	0	992	670	0	0	0
primary	0	0	0	0	992	670	0	0	0
secondary	0	0	0	0	938	639	0	0	0
Component Elements									
chapter	4	3	8	2	2	2	6	4	7
appendix	6	0	0	0	1	1	2	2	1
Section Elements									
sect1	22	18	28	16	30	20	21	34	31
sect2	19	13	19	3	123	39	15	12	27
sect3	14	0	9	0	14	0	1	0	12
sect4	0	0	0	0	5	0	0	0	2
sect5	0	0	0	0	0	0	0	0	10
bridgehead	0	3	0	0	580	0	0	0	0
Meta-Information Elements									
author	0	0	0	0	0	0	0	0	0
beginpage	5	0	0	1	0	0	0	0	0
biblioid	1	1	1	1	1	1	1	1	1
bookinfo	1	1	1	1	1	1	1	1	1
city	1	0	0	0	1	1	0	0	0
country	1	0	0	0	1	1	0	0	0
corpauthor	1	1	1	1	1	1	1	1	1
date	9	10	4	4	11	10	11	7	3
email	1	0	0	0	1	1	0	0	0
keyword	14	14	10	10	14	14	0	0	14
keywordset	1	1	1	1	1	1	0	0	1
postcode	1	0	0	0	1	1	0	0	0
productname	1	1	1	1	1	1	1	0	1
pubsnumber	0	0	0	0	0	0	0	0	0
revhistory	1	1	1	1	1	1	1	1	1
revision	9	10	4	4	11	10	11	7	3
revnumber	9	10	4	4	11	10	11	7	3
revremark	9	9	4	4	9	10	10	7	3

Element	Vol. 1	Vol. 2	V2 Sup1	V2 Sup2	Vol. 3	Vol. 4	Vol. 5	Ra-tionale	IHB
subtitle	1	1	1	1	47	1	1	1	1
title	75	47	102	24	195	68	59	62	115
volumenum	1	1	1	1	1	1	1	1	1
List Elements									
itemizedlist	18	16	26	9	226	19	10	17	24
orderedlist	1	2	4	0	20	0	0	9	1
listitem	90	86	121	22	1110	95	47	86	132
term	2	0	0	0	0	0	0	0	8
variablelist	1	0	0	0	0	0	0	0	1
varlistentry	2	0	0	0	0	0	0	0	8
Block Elements									
address	1	0	0	0	1	1	0	0	1
para	242	182	338	136	2213	231	180	414	348
programlisting	0	0	0	0	0	0	0	0	0
remark	0	0	0	0	304	1	1	0	0
Inline Elements									
citation	0	1	0	0	0	0	0	0	0
command	0	0	0	0	1	0	0	0	0
emphasis	30	45	48	31	142	373	155	167	6
footnote	9	4	0	17	4	4	16	6	0
footnoteref	0	0	0	0	0	0	10	0	0
link	0	0	0	0	0	0	0	0	0
literal	0	0	0	0	0	0	0	0	0
superscript	0	0	0	0	1	0	0	0	0
ulink	7	0	0	0	347	0	0	0	1
xref	15	6	18	2	17	18	6	14	0
Graphic Elements									
caption	0	0	0	0	0	0	0	0	0
figure	8	11	32	2	14	3	6	5	21
imageobject	16	22	64	4	106	6	12	10	42
imagedata	16	22	64	4	106	6	12	10	42
informalfigure	0	0	0	0	0	0	0	0	0
mediadata	0	0	0	0	0	0	0	0	0
mediaobject	8	11	32	2	53	3	6	5	21

Element	Vol. 1	Vol. 2	V2 Sup1	V2 Sup2	Vol. 3	Vol. 4	Vol. 5	Ra-tionale	IHB
Table Elements									
colspec	8	5	22	0	1216	76	29	23	12
entry	96	20	165	0	5680	2242	604	3023	42
informaltable	2	0	0	0	563	12	0	1	1
row	48	4	42	0	2751	396	128	744	14
table	1	1	5	0	4	1	6	4	2
tbody	3	1	5	0	567	13	6	5	3
tgroup	3	1	5	0	567	13	6	5	3
thead	1	0	4	0	28	13	5	5	3

Any DTD describes a number of elements and the relationship between these documents, however there is no syntactic construct, which identify a single element at the default root elements. Often a written description of the DTD identify a such a root element, however this is not the case for the docbook DTD. Theoretically any element can be a root element in DocBook, but in practice the root element of most DocBook documents are either the `article` or the `book` element.

All the documents in the do have the `book` element as its root element.

5.3. DocBook Markup in the NISP

The following section illustrates how to use a small subset of the elements defined in DocBook. For each of these elements only the necessary attributes are included. For a much more comprensive definition, see the reference manual [WaMu05]. The elements selected illustrates common constructs in word processin in general, and covers almost all the elements required to write the documents.

The documents are all created with the DocBook element *book* as its root element. The book element consists of meta-information about the book in form of the *bookinfo* element, a number of chapters, and potentially a bibliography element and a number of appendixes, as shown below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<book>

<bookinfo>
  <title>NC3 Technical Architecture</title>
  <subtitle>NCOE </subtitle>
```

```
  ...
  <volumenum>5</volumenum>
  ...
  <revhistory>
    <revision>
      <revnumber>&src-version;</revnumber>
      <date>&src-release-date;</date>
    </revision>
    <revision>
      <revnumber>3.0</revnumber>
      <date>15 December 2001</date>
    </revision>
  ...
  </revhistory>
</bookinfo>

<chapter><title>Some title</title>

<sect1><title>My first section</title>

<para> Some text</para>

<sect2><title>A subsection</title>

<para>Some more text</para>

</sect2>

</sect1>

</chapter>

  ...

<appendix><title>Some other title</title>

</appendix>
</book>
```

The meta-info element *bookinfo* contains among other things a *revhistory* element, which consists of a list of *revision* elements. In the draft version 4 of the , the first *revision* element uses external entities, which are defined in the file `VERSION` located in the `src` directory. When the final version of volume 4 is released, the `VERSION` file should be updated accordingly. But during the development, he user could create his own revision element.

Each chapter and appendix consists of a number of elements, where the most significant are *para* and *sect1* which constitutes a paragraph respectively a section on level 1.

Each *sect1* element consists of a number of elements, among others the *sect2* element etc.

The first element in all chapter-, appendix-, and sect1- to sect4 element is by convention in the a title element.

Just as the *book* element contains meta-info through the *bookinfo* element, there are similar elements for the *chapter*, *appendix* and the section elements. These elements are currently not used, but might be in the future.

5.3.1. Creating lists

DocBook supports seven list elements. Two of those are:

ItemizdList	An unordered (bulleted) list. There are attributes to control the marks used
OrderedList	A numbered list. There are attributes to control the marks used

The following examples illustrated these two list types:

A bullet list, with the DocBook source:

```
<itemizedlist>
  <listitem><para>First Item</para></listitem>
  <listitem><para>Second Item</para></listitem>
  <listitem><para>Third Item</para></listitem>
</itemizedlist>
```

looks like this:

- First Item
- Second Item
- Third Item

And an ordered list, with the DocBook source:

```
<orderedlist>
```

```
  <listitem><para>First Item</para></listitem>
  <listitem><para>Second Item</para></listitem>
  <listitem><para>Third Item</para></listitem>
</orderedlist>
```

looks like this:

1. First Item
2. Second Item
3. Third Item

Lists can also be nested, and bulleted and numbered list can be combined arbitrarily, as illustrated by the source:

```
<itemizedlist>
  <listitem><para>First Item</para></listitem>
  <listitem><para>Second Item</para></listitem>
  <listitem><para>Third Item
    <orderedlist>
      <listitem><para>First Item (2nd level)</para></listitem>
      <listitem><para>Second Item (2nd level)</para></listitem>
      <listitem><para>Third Item (2nd level)</para></listitem>
    </orderedlist></para></listitem>
</itemizedlist>
```

looks like this

- First Item
- Second Item
- Third Item
  1. First Item (2nd level)
  2. Second Item (2nd level)
  3. Third Item (2nd level)

Instead of using numbers in an ordered list, we can use letter, roman numeral all controlled with the attribute *numeration*.

5.3.2. Typographical element

A few elements are oriented toward the *What You See Is What You Get* solution, familiar from well known word processors. Among the elements are:

- Italic text like *Hello Italic World*, is created using the following markup:

```
<emphasis>Hello Italic World</emphasis>
```

- Bold text like **Hello Bold World**, is created using the following markup:

```
<emphasis role="bold">Hello Bold World</emphasis>
```

- Subscripted text like H_2O, is created using the following DocBook markup:

```
H<subscript>2</subscript>O
```

- Superscripted text at the square function: X^2, is created using the following DocBook markup:

```
X<superscript>2</superscript>
```

5.3.3. Creating Footnotes

Footnotes are generated using the following DocBook markup: `<footnote><para>This is my footnote</para><footnote>`. In practise this will result in the following[2]

What actually happens here is twofold. The footnote is generated and will later be added on the bottom of the current page, and at the same time a reference to the same footnote is generated. Sometimes one needs to referrer to the same footnote from multiple place. The can be accomplished by creating an id attribute to the footnote like `<footnote id='xray'><para>This is my footnote</para><footnote>`. We can then later refer to the same footnote[2], using the markup `<footnoteref linkend='xray'/>`.

5.3.4. Graphic Elements

Graphics occurs often in Figures (the elements *figure* and *informalfigure*) and Screenshots (the element *schreenshot*), but they can also occur without a wrapper. The element used to contain graphics and other media types are the elements *mediaobject* and *inlinemediaobject*[3]. These elements may contain video, audio, image and text data. A single *mediaobject* may contain multiple version, and it is up to the presentation system to select the appropriate object. The respective elements contained in the *mediaobject* element are the elements *videoobject*, *audioobject*, *imageobject* and *textobject*.

The following example illustrates[4] the use of a graphic elements, in the way it is often used in the documents:

```
<figure><title>Here is an image</title>
  <mediaobject>
    <imageobject>
      <imagedata fileref="figures/a-raster-image.svg"/>
    </imageobject>
  </mediaobject>
</figure>
```

In a lot of the tables, in appendix A of volume 3 we actually use the *mediaobject* directly, since the *figure* element is not part of the table cell (*entry*) content model.

In version 2 of the all volumes used different and inconsistent naming conventions for the raster images used in the HTML version. The figures, was named Figure01, Figure001, etc. all names, more related to the figure number in the work document, which

[2]This is my footnote
[3]Although still legal, the element *graphics* and *inlinegraphic* will be removed in a future version of DocBook.
[4]In the , we actually only use entity references as value to the attribute *fileref*.

by the way was not always correct.

Since all the figures during the translation of the to XML, was implemented as Scalable Vector Graphics (SVG) files, a new naming scheme was implemented at the same time.The naming convention is implemented in the following way:

- All figures are places in a separate figures directory.
- A figure could eg be named *figures/v1-ex-mgt.svg*. This describes the directory, where the figure is located and the name of the SVG image representing the figure.
- Note, that figure names consists of the parts *v1*, *ex-mgt*, and as extension svg, where
  - *v1* is the volume the figure is located in
  - *ex-mgt* is a shortcut for the title of the figure
  - and svg is the extension of the file

5.3.5. Creating Tables

Table is a significant element of the NISP. In version 3, we currently have 502 informal- and formal tables. The difference between the two elements is that *table* have a caption and *informaltable* does not.

Most tables are actually straightforward to create, as should be familiar to everybody, which have worked with HTML files, since the DocBook table model is very similar to the CALS model used in HTML.

In DocBook the *table* element consists of a number of *row* element. Each *row* consists of a number of *column* elements, which are specified by the *cols* attribute of the *tgroup* element as illustrated below:

```
<table frame="all">
<title>Simple Table</title>
<tgroup cols="3">
<colspec colwidth="33*"/>
<colspec colwidth="33*"/>
<colspec colwidth="34*"/>
<tbody>
  <row>
    <entry>A</entry>
    <entry>B</entry>
    <entry>C</entry>
  </row>
```

```
  <row>
    <entry>D</entry>
  </row>
</tbody>
</tgroup>
</table>
```

resulting in:

A	B	C
D		

Table 5.1. Simple Table

Note, that in the second row, only one cell have been specified, which illustrates that the stylesheet assumes an entry to be empty, if it isn't specified. The processing of cells in a row is done from left to right, and each cell meet is assumed to be the next cell in the row, unless explicit specified in an attribute. So if we define that a table have 5 columns, but only create 2 entries without any position modifier. These cells are assummed to be the first and second cell in a row. The three other cells are considered empty.

The next example illustrates the concept of cells,which span multiple columns and/or rows. The *"Protocol Set"* part of most of the tables in Appendix A of Volume 3 utilises this facility, which is illustrated below:

```
<table frame="all">
<title>Complicated Table</title>
<tgroup cols="3">
<colspec colname="c1" colwidth="33*"/>
<colspec colname="c2" colwidth="33*"/>
<colspec colname="c3" colwidth="34*"/>
<tbody>
  <row>
    <entry>A</entry>
    <entry namest="c2" nameend="c3">BC</entry>
  </row>
  <row>
    <entry morerows="1">DG</entry>
    <entry>E</entry>
    <entry>F</entry>
  </row>
  <row>
```

```
    <entry>H</entry>
    <entry>I</entry>
  </row>
</tbody>
</tgroup>
</table>
```

resulting in:

A	BC	
DG	E	F
	H	I

Table 5.2. Complicated Table

5.3.6. Creating URLs

Links, that addresses its targets by means of a URL are implemented through the *ulink* element. A typical use of the element, would be:

```
<ulink url="http://www.nc3a.nato,int">NATO C3 Agency</ulink>
```

which would look like this "NATO C3 Agency [http://www.nc3a.nato,int]". Sometimes we want the presented text (here "NATO C3 Agency") to be the URL itself. This can be accomplished with the following abbreviation, which describes an element with no contents, such as:

```
<ulink url="http://www.nc3a.nato,int"/>
```

which would look like this: http://www.nc3a.nato,int

5.3.7. Creating Cross references

Creating cross-references in DocBook is quite easy. Every element may have an attribute *id*, which must be set to a unique value. In order to reference the element anywhere in the document, we create an *xref* element, which then will refere to the ele-

ment. The text used in the reference is dependent on the context. So a reference to a chapter would e.g. be *Chapter 3*, and similar for other elements.

```
<xref linkend="v4-ch-my-intro"/>
```

In the documents, we by convention uses id's, that are identifiable. like eg. *v4-ch-intro*. The user should only create is in the different elements, if there is a requirement for to reference that element.

6. Design of the TA Stylesheets

XML documents are usually transformed into a format suitable to most readers. For the TA documents, the current choices of output medias, is either in the form of HTML[1] pages suitable to be displayed by a web browser, or in a form suitable for output on regular paper, which in our case will be PDF, which both can be viewed on screen, og printed out on paper.

The best tool for transforming XML documents is the XML Stylesheet language (XSL). The XSL standard consists of two parts "the XSL Transformation" (XSLT) [XSLT]and the XSL formatting object language (XSL-FO) [XSL], the first being a tool to transform XML documents, and the later is a highly complicated formatting language.

6.1. Basic XSLT

An XSLT stylesheet consists of a number of templates, which describes, how to transform the different elements from XML to another language like HTML, or potentially another XML language. XSLT is also a declarative XML language, with a number of pre-defined elements, which defines how to handle XML elements, common programming language constructs like loops and conditional statements etc. XSLT utilises the XPath standard [XPath] to define the patterns, which describes elements of the source language.

Figure 6.1. Applying XSLT to an XML document

There will be no further description of XSLT in this document. Instead, the user are can read the introductory article [Ho00]. For a comprehensive information on XSLT, see the excelent book[Kay01] written by Michael Kay, the current editor of XSLT.

[1]We are actaully creating XHTML pages, which is an XML implementation of HTML

6.2. The DocBook Stylesheets

Creating stylesheets for a large DTD as the DocBook DTD is a huge task. Fortunately the chairman of OASIS technical committee Norman Walsh have implemented a collection of XSLT stylesheets, which enables the user to transform from DocBook to HTML and XSL-FO. This stylesheet collection is is not without reason considered one of the most complicated implementations of XSLT yet devised.

The implementation of the stylesheets are based on a number of design criterias, of which the most important is the decision to make the stylesheets highly modular and parameterised. Furthermore the stylesheets are highly data-driven, which is accomplished by using matching templates as much as possible.

Together, these and additional criterias makes it reasonable easy to design customisation of the stylesheets. However, some of the functionality of the DocBook stylesheets, can not or only with great difficulty be implemented with XSLT. For this purpose a number of small Java classes have been created, which works as an extension to the XSLT language. In order for these extensions to work, a parameter must be set in the stylesheets, and a Java jar file must be included on the classpath. Since we are using the Saxon XSLT-processor, we need to add the appropriate Saxon extension to the Java classpath[2] and set the parameter `use.extensions` in the stylesheet. Besides the official reference material included in the DocBook XSL distribution, a draft book [St03] is currently availible, which in detail describes how to use the DocBook XSL stylesheets.

The stylesheets used to create the are customised version of the XSLT stylesheets, created to comply with the design of the TA documents. None of the original DocBook XSLT stylesheets are modified in any way, instead the stylesheets imports the DocBook XSLT stylesheets and through parameterisation and overriding of templates, the behaviour of the stylesheets are modified in order to fit with our design requirements.

6.3. TA Stylesheets for HTML

The currently consists of volume 1 to 5, the rationale document and a couple of suplements, all in the form of XML files. Besides these files, a number of files exists in the `src/master` firectory , which describes three pages: one is the front page shown in the web version of the and the other two pages are both available from the front page through the links *User Information* and *Introduction*. All of the volumes are in the XHTML version being displayed as chunked pages.

The current XSLT standard does not support the creation of multiple target documents, but a now depreciated working draft of version 1.1 of the standard, and more recently a working draft[XSLT2] of what is to become version 2 of the XSLT standard, support this and several other additions, and are already being supported by most XSLT processors, although sometimes through a proprietary extension. The Saxon

[2]The Saxon extension is located in the directory `xsl/docbook-xsl/extension/`

XSLT processor used in the , have the advantage of being developed by the current editor of the XSLT document and are therefore usually way ahead of the competition. The - and the master pages all uses the same layout as described in the following figure.

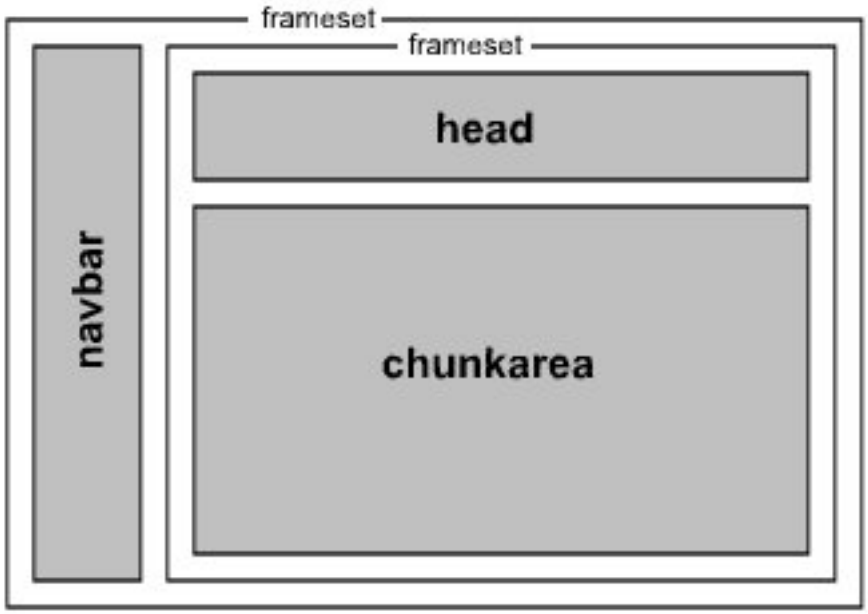

Figure 6.2. Generic Frame Layout

The gray areas represent the frames with actual contents, while the two other rectangles are just containers in the form of *frameset* definitions as described in [Mu02]. Each of the three frames are given a name as illustrated in Figure 6.2, “Generic Frame Layout”, which then are referenced to using the attribute *target* of the HTML anchor element *a*. The generated code for the layout of frames are shown below.

```
<html>
  <frameset framespacing="0" frameborder="0" cols="140,*">
    <frame src="navbar.html" name="navbar" marginwidth="0"
           marginheight="0" framespacing="0" frameborder="0" border
    <frameset framespacing="0" frameborder="0" rows="95,*">
      <frame src="head.html" name="head" noresize scrolling="no">
      <frame src="firstpage.html" name="chunkarea" scrolling="auto"
    </frameset>
  </frameset>
</html>
```

The above frame layout code is used in all documents and also for the master docu-

ment forming the frontend to the web portal. This layoutcode will be saved in a file named `index.html` for each document in a seperate directory. Besides this file, two additional files will also be creates for the frame called *navbar* and *head* called respectively `navbar.html` and *head.html*. The frame called *chunkarea* will be used for a chunked version of the document, where the first chunk will be named `first-page.html`.

The stylesheet *multipage.xsl* will create all these file and will also use a number of cascaded stylesheets [Li99] for detailed formatting of the different HTML elements.

6.4. TA Stylesheets for XSL-FO

The page description model used by XSL-FO is fundamentally different from the HTML model and is used to very precisely describe the layout of a page. An XSL-FO document describes how a series of nested boxes are placed on one or more pages. A XSL-FO document contains a layout-master-set and one or more page-sequences. The layout-master-set contains a number of page-masters, which each describes the physical layout of a page. The pages created from a page-sequence are also based on one or more page-masters from the layout-master set. The final process, after creating an XSL-FO document is either to view the document using an XSL-FO viewer or transform it to another format. We do the later with the document, and transform to the page description language PDF.

Conceptually the DocBook XSL-FO stylesheets by default groups a DocBook document up into six predefined classes and associates a page-sequence-master to each group. Each of these page-sequence-masters can be formattet differently. Within each page-sequence-master we defined typically four different page-master, describbing the layout of a of a page. Typically, we have different page-masters for the first, odd, even and blank pages in a page-sequence.

Page Class (and page-sequence master names)	**Used by**
titlepage	Set-, part- and Book titlepages
lot	List of titles including table of contents and list of figures, tables and examples

front	Used by these elements:
	
	`dedication`
	`preface`

body	Used by these elements
back	
index	

Table 6.1. DocBook XSL-FO Page Classes

6.4.1. Layout of Pages

All pages in a XSL-FO document will have the same basic layout, as illustrated in Figure 6.3, “Layout of a XSL-FO page”. For each of the letters A-H, there exists a parameter in the stylesheets, which can be defined. The NISP stylesheet `xsl/fo/ta2fo.xsl` set these parameters in accordance with the design of the original Word documents.

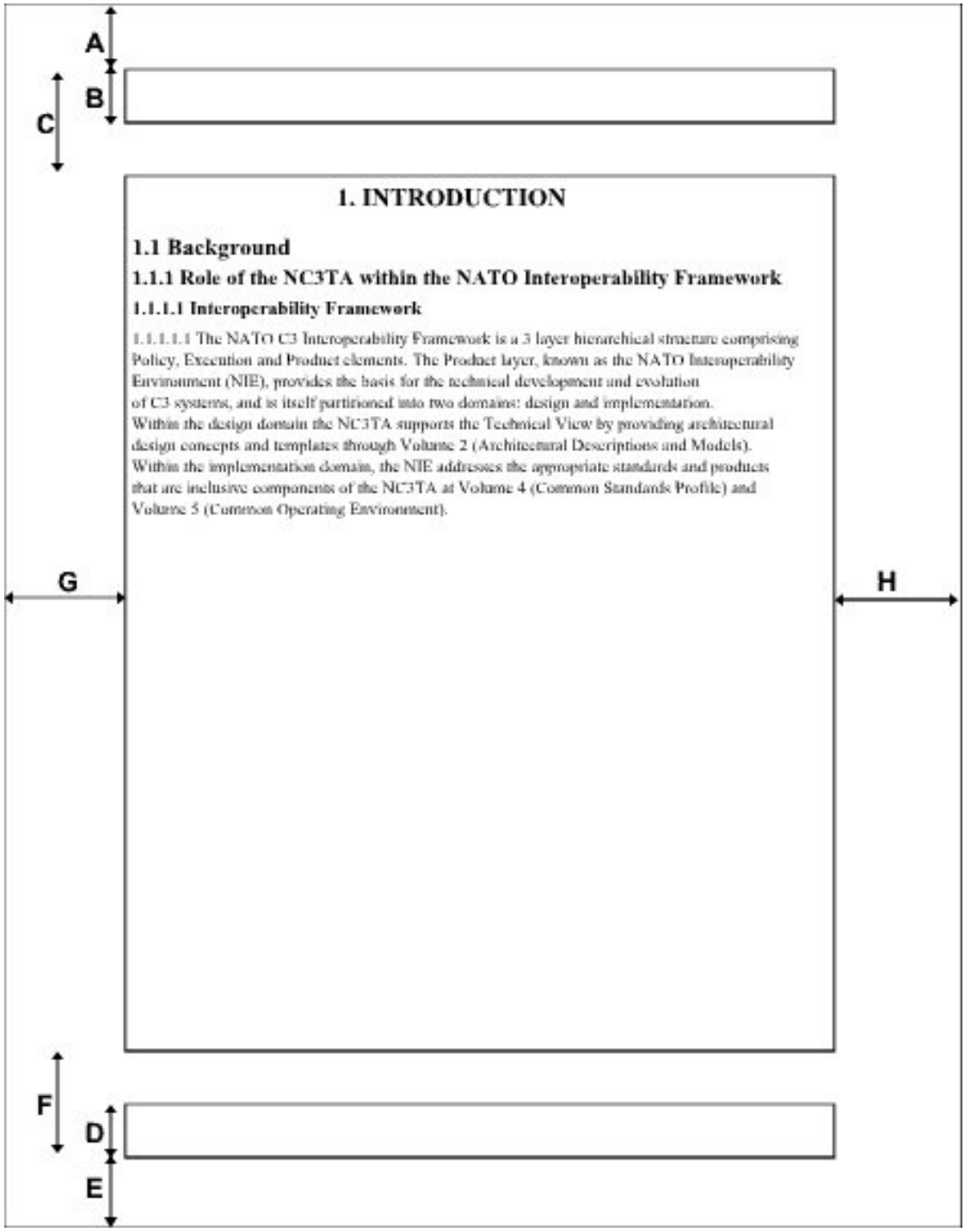

Figure 6.3. Layout of a XSL-FO page

Many of the layout parameters are customisable through simple parameters defined in the stylesheet. And most of the layout and design can actually be changed, by just

changing the value of a parameter. However, some of the templates are not suitable for the design of the , and must therefore be overwritten accordingly. Other templates are modified either because bug in the Docbook stylesheets, or due to unimplemented features in the XSL-FO processor used subsequently in the production of PDF files.

6.4.2. Titlepage templates

Some things can not easily be accomplished through parameterisation, e.g the titlepage of a document. For these purposes, a special template system have been implemented in the DocBook XSL stylesheets. The templates defines the layout of the titlepage, and also the format of *titlepages* of the major component (chapter, appendixes and sections), which means where and how the different components should be formatted, selection of fonts, size of fonts, etc.. The stylesheet used to create the XSL-FO documents have the name `xsl/fo/ta2fo.xsl`. The file `xsl/fo/nisp-layout.xsl` contains the styles used to modify the *tilepages*[3]

6.4.3. Other FO stylesheets

Additional stylesheets which are used to create the XSL-FO of the document are:

- fo-post-for-fop.xsl - Used to post-process the XSL-FO result tree. This enables us to temporary fix FOP bugs in a simpler way, than modifying the DocBook XSLT stylesheets.

[3]This stylesheet is created from the specification defined in `xsl/fo/nisp-layout.xml`

7. Working with the

Due to the requirement of creating both a HTML- and a PDF-version of all the documents. The translation all the architecture documents using the XML- and SVG files, the XSLT stylesheets and tools described in this document is quite a complicated operation, which requires that the user executes dozens of commands in the correct order, not to mention the requirements when creating a distribution of all the translated documents for the web. To simplify this operation, a make-like tool called Ant is used, and acts as a kind of glue, between the different tools required in the publication cycle. The Ant tools are used throughout the publication cycle process except when editing the NISP documents, and are also used by the editor and tool package maintainer to create new distributions of the source- and tool packages.

The following sections will describe the content of the source distribution[1], and describe all the steps necessarry in order to create human readable versions of the architecture documents.

7.1. Inside the Source Distribution

The sources to the is distributed separately from the tools distribution in a compressed archive. The name of the latest distribution is `nisp-src-3.0.zip` and is organised in the following way:

7.1.1. Structure of distribution

The source distribution consists of the following elements:

- a number of directories, one for each of the documents in the distribution (`volume1`, `volume2`, `vol2-sup1`, `vol2-sup2`, `volume3`, `volume4`, `volume5`, `rationale` and `ihb`), where the `ihb` directory should contain pdf versions of the *Implementation HandBook*.
- The directory `schema`, which contains all the schemas used in the documents.
- The directory `acronyms` contains a list of acronyms created by NC3A. This is only included in the HTML version.
- The directory `master` contains a number of DocBook files in form of chapters a consisting of four *chapters*. Each *chapter* represents a HTML page, which are used to create misc. pages for the web portal.
- The directory `olinkdb.xml` contains a database of document references for all the documents. The database contains pointers in the form of external entities,

[1]Where all the editing takes place.

which point to an auto generated list of reference for each of the volumes.

- The directory `standards`, which contains the combined collection of standards used in chapter 2 of volume 3 organised in a hierarchical database-like structure. This database is also used by volume 4 (NCSP) and the rationale document.

- The file `catalog.xml`, which is a XML Catalog used to resolve public identifiers in the documents.

7.1.2. Database driven documents

In the original word version of volume 3 (Base Standards), the description of each of the standards were presented in a similar fashion, in the form of a table. During the transformation from the Word version to XML, all the standards were extracted from chapter 2 in volume 3 and placed in a separate XML document, which is organised like a hierarchical database. This was done for two reasons, primary because a table have no semantic value and secondary because, it was a nuisance to translate a couple of hundred word tables to similar XML tables. When the translation of volume 3 to HTML and FO takes place, a merged version volume 3 is created using processing instructions as described in Section 4.1.4, “Processing Instructions”, i.e. the database `standards/ta-standards.xml` is merged back into the document[2], which is then translated to HTML and FO as illustrated in Figure 7.1, “Database driven documents”. As of version 4 of the , both volume 4 and the rationale document uses a similar approach. In this way, we will avoid the problem of selecting standards for the NCSP, which have not been defined in volume 3.

Figure 7.1. Database driven documents

7.1.3. The standards database

[2]Currently volume 3 consists of the skeleton files vol3.xml, and v3appendixA.xml

The standard database was designed to mirror the original document structure, where the chapter was split into eight service areas. Each service area was grouped into miscellaneous service classes, which again contained a list of standards. The following code illustrates this heirachical structure, and also illustrates the structure of standards. The syntax of standards is defined in the file `schema/dtd/stddb.dtd`, and are also included in this document in Chapter 2, *The standard database DTD*, with a more detailed descrition of the DTD.

```
<ta-standards>
  <servicearea id="se" title="Software Engineering Services">
    <serviceclass cid="se-lng" title="Languages and Bindings">

      <standardrecord id="se-lng-ada95" tag="Ada">
        <standard orgid="&iso-iec;" pubnum="8652" date="1995"
                  title="Ada">
          <correction cpubnum="TC1" date="2001"/>
        </standard>

        <applicability>Ada is a general purpose high-order
        object-oriented programming language capable of processing
        both numerical and textual data that has the key attributes
        of strong data typing, data abstraction, object-orientation
        structured constructs, multitasking and concurrent
        processing. For R&amp;amp;D it is encouraged to use Ada95
        </applicability>

        <status stage="30.40">
          <history>
            <event flag="added" date="1998-11-06" version="1.0"/>
            <event flag="changed" date="2003-08-01" rfcp="" version
          </history>
        </status>
      </standardrecord>
    </serviceclass>
  </servicearea>
</ta-standards>
```

In order to merge the standards back into volume 3, we have in the documents created handles in the form of processing instructions (PIs), which are then substituted with the appropriate standard, when creating volume the final version of volume 3. The PI defined in the skeleton version of volume 3, do not referer to a specific standard, but rather to a *serviceclass*, A typical PI used to e.g. include the *Language and Bindings* service class, look like:

```
  <?dbmerge cid="se-lng"?>
```

7.1.4. DTD

The directory `schema/dtd` contains all the DTDs used in the documents. The most important is of course the ver %dtd-ver; of DocBook XML DTD used to develop the document.

- `acronyms.dtd` - describes the syntax of the acronyms list
- `docbkx42` - This directory contains the all the files describing the syntax of DocBook documents and which is used by all documents in the technical architecture.
- `rfcp.dtd` - contains an experimental DTD for Request For Change Proposals (RFCP). The DTD may describe documents, which can be created dynamically by submitters of the RFCPs.
- `stddb.dtd` - describes the standards included in volume 3, and also a view of selected mandatory- and emerging standards, which are shown in the standard class tables in volume 4 and in the traceabillity matrix included in the rationale document. The `stddb.dtd` is a customized version of the DocBook XML 4.3 DTD. Since several of the elements contains DocBook code, which should passed-on unchanged to the document requiring the definition standard, it is much easier to create an extension to DocBook, rather than working with multiple namespaces. Additional element have recently been added to the DTD to enable an autogeneration of tables in volume 4 and the rationale document as well. The standard database is described in more detail in Chapter 2, *The standard database DTD*
- `svg10.dtd` - This is the official DTD describing Scalable Vector Graphics documents.
- `targetdatabase.dtd` - This DTD describes a target database, used to create cross-references between documents.

7.1.5. Figures

The directory `figures` contains an SVG implementation for each of the images used in the documents.

7.2. Editing the

In principle, all the XML/XSLT and SVG files in the source distribution can be edited using the users editor of choice. In reality, this is not always the case, since not all editors by default supports editing of Unicode documents yet. Most English documents written in the Unicode standard, which is a 31 bit character set, generally uses an encoding scheme, which allows most document to backward compatible with ASCII. All the XML documents forming the technical architecture documents are created using

an encoding scheme called ISO-8859-1 (also called Latin 1), which by default is support by the XML standard. Modern versions of Word supports various Unicode encoding schemes, although it is not obvious from the user interface, what encoding schemes are actually supported. In one version it was stated that word was using Unicode, but what encoding was not specified. This is a pity, since Microsofts support for Unicode in general is quite good. Furthermore when copying text between word document and text documents, Word usually copy additionally *invisible characters*, which XML parsers don't like. For this and additional reasons an open source editor jEdit (http://www.jedit.org/) is included with the tool distribution. Every contributor to the documents are *STRONGLY ENCOURAGED* to use this editor, which have the additional advantages of being XML- and DocBook aware and being able to integrate the tools package used to build the documents.

7.3. Processing the

7.3.1. The Build Tool Ant

The Ant tool developed by the Apache group, build its targets[3] according to an XML based configuration file. In the tools distribution we have two Ant configuration files. One is called `build.xml` and is used to generate the architecture document for nations and the NATO intranet are all these document are thefore labeled *NATO/ EAPC UNCLASSIFIED*. The other file is called `build-internet.xml` and is used to create a version of the with will be published on the internet and are thefore created without any classification labels. Both files are located in the root directory of this distribution. These build files are more or less self-explanatory, and describes the different targets and sub-targets, which should be executed in order to create HTML- and PDF versions of the . If some of the XML files and XSLT stylesheets are modified after a build, only the targets, which depends on the modified files are build in a subsequent execution of Ant.

This build files also takes care of the creation of new distributions, as e.g. a new nisp tools distribution, a new distribution of the source and also takes care of the task of creating a web version of the technical architecture, etc.

7.3.2. Running Ant from a Command Shell

As all Java programs in the tool distribution, the Ant executable is embedded inside a Java jar file. In order to simplify the execution, two script files `build.bat` and `build.sh` are included for Windows respectively Unix. These scripts calls the Ant program and is executed using the syntax:

```
.\build.bat [options] [target [target2] ...]]  (Win32)

./build.sh  [options] [target [target2] ...]]  (Unix)
```

[3]A target is a task, e.g. translate volume 1 to HTML

In order to run Ant on the provided build file, the user should open a command shell, and then run the script files from the directory, where the build file is located. The legal parameters for ant and therefore the build script are the following:

```
build [options] [target [target2 [target3] ...]]
Options:
  -help                  print this message
  -projecthelp           print project help information
  -version               print the version information and exit
  -diagnostics           print information that might be helpful to
                         diagnose or report problems.
  -quiet, -q             be extra quiet
  -verbose, -v           be extra verbose
  -debug                 print debugging information
  -emacs                 produce logging information without adornm
  -logfile <file>        use given file for log
    -l     <file>                ''
  -logger <classname>    the class which is to perform logging
  -listener <classname>  add an instance of class as a project list
  -buildfile <file>      use given buildfile
    -file    <file>              ''
    -f       <file>              ''
  -D<property>=<value>   use value for given property
  -propertyfile <name>   load all properties from file with -D
                         properties taking precedence
  -inputhandler <class>  the class which will handle input requests
  -find <file>           search for buildfile towards the root of t
                         filesystem and use it
```

If the name of the actual build file is not specified in the example, the ant system will by default select the build file `build.xml`. So in order to create a classified version of the , In the above example, the following command should be executed:

```
.\build.bat -f build-internet.xml [options] [target [target2] ...

./build.sh  -f build-internet.xml [options] [target [target2] ...
```

7.3.3. Running Ant from jEdit

When using the editor jEdit, it is now possible to edit the documents and build all the

html- and pdf versions of the from within jEdit. The jEdit plugin AntFarm integrates the Ant tool, and thereby the building process into the editor.

FIXME:

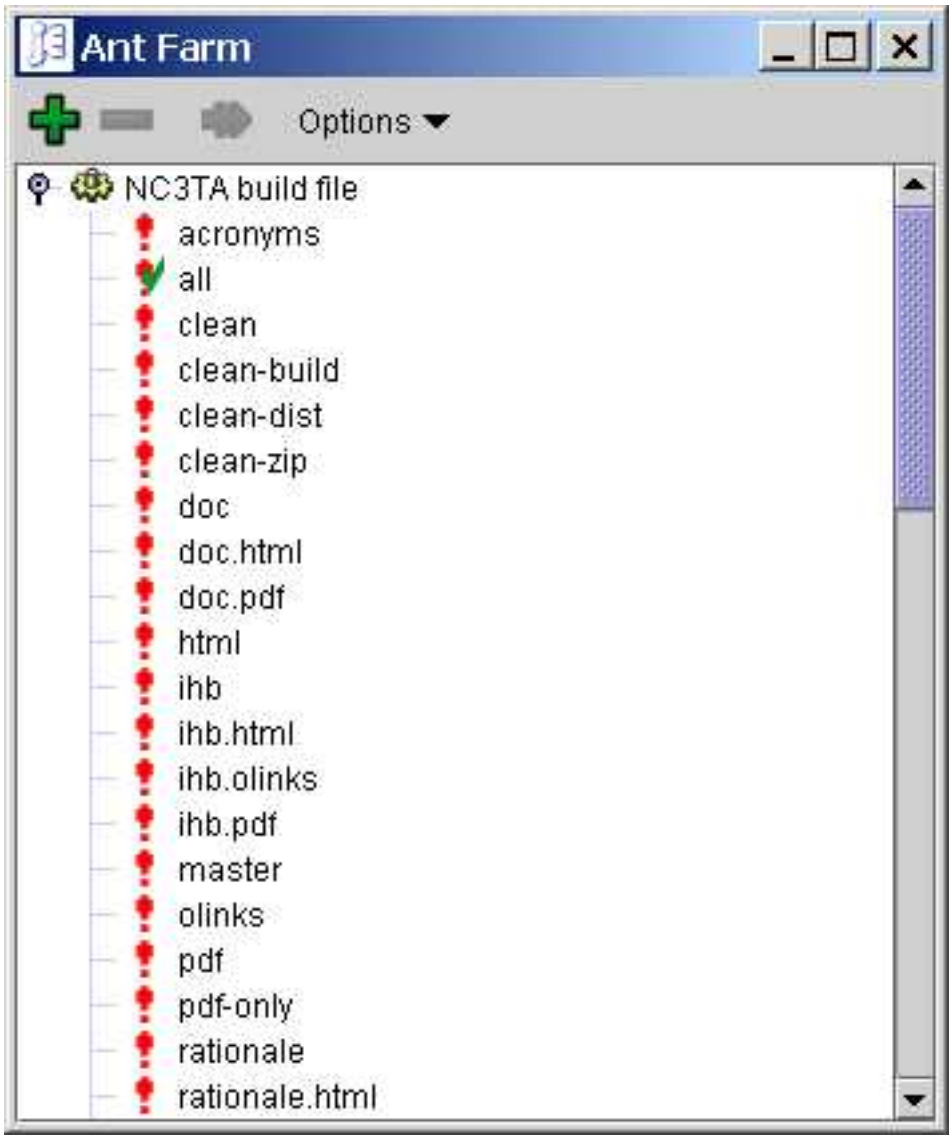

Figure 7.2. AntFarm Build

7.3.4. General Ant Targets in build.xml

The build file `bild.xml` comes with a number of predefined target, which are goals, that must be solved. On such goal could be to create a chunked HTML version of a volume, another goal could be to create a PDF version of a volume.

The dependency of a number of the targets are illustrated in Figure 7.3, “Target Dependencies”. There we can see, that the default target *all* depends on the targets *html* and *pdf*. The target *pdf* depends on *vol1.pdf*, *vol2.pdf* etc. Note that *vol1.pdf* is a target and have nothing to do a file with a similar name.

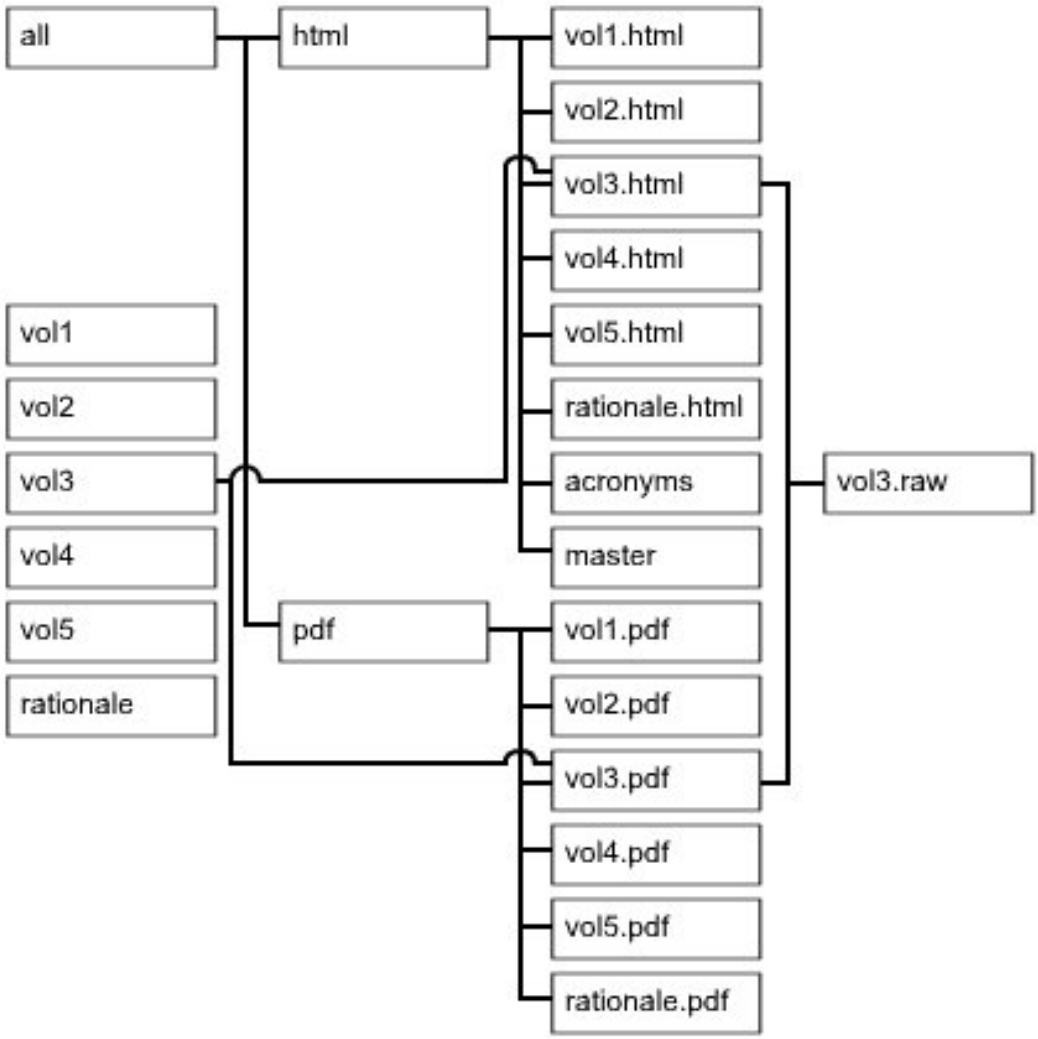

Figure 7.3. Target Dependencies

Currently the main targets are:

Volume 1	Volume 2	Volume 3	Volume 4	Volume 5	Rationale Document	Comment
vol1	vol2	vol3	vol4	vol5	rationale	Create a

Volume 1	Volume 2	Volume 3	Volume 4	Volume 5	Rationale Document	Comment
						HTML and PDF version of the document
vol1.html	vol2.html	vol3.html	vol4.html	vol5.html	rationale.html	Create a HTML version of the document
vol1.pdf	vol2.pdf	vol3.pdf	vol4.pdf	vol5.pdf	rationale.pdf	Create a PDF version of the volume
-	-	v3.merge[a]	v4.merge	-	rd.merge	Merge the standards into the volume
valid1	valid2	valid3	valid4	valid5	valid-rationale	Check if the document is valid[b]

[a]Merge the standard database with volume 3. What it actually does, is to merge the content of the files `vol3.xml`, `v3appendixA.xml` and `ta-standards.xml` and as a result create the file `vol3-merged.xml`

[b]Note the jEdit editor automatically validates a document, if the XML plugin is installed

There is actually an additional target for volume 3, which is called `vol3-check`, but this target is used by one of the targets already mentioned to decide if it is necessary at all to rebuild a given target, depending on the date of the files involved. This target should therefore never explicit be executed by the user. For a list of all the targets available, the user should under windows run the command:

```
.\build.bat  -projecthelp
```

Besides similar targets for the other volumes, and targets for the master document and acronyms a few targets should also be mentioned.

Target	Comment
all	If the user do not supply a target, the default target *all* is build. This target create a `build` directory[a], and then build all volumes, and also copy additionally files to this directory, such as e.g. the figures and icons used in the documents, plus additional files necessary.
html	Create HTML of all documents, the rationale, the master and the acronyms pages.

Target	Comment
pdf	Create a PDF version of all the documents, and the rationale document.
doc	Create the documentation of the tools distribution. That is currently only this document.

[a]This directory is located in the root of the tools distribution

A number of target exists specifically to clean up the filesystem of generated. Usually these targets deletes the generated html and pdf files.

Target	Comment
clean-build	Delete the `build` directory.
clean-dist	Delete the `dist` directory
clean-zip	Delete the `zip` directory
clean	Remove all generated files, from the `build`- and `dist` - directories.

7.3.4.1. Special distribution targets

The following targets, should only be used by the editor of the and the maintainer of the nisp tool package. All the archives generated by these targets are saved in the sub-directory `zip`.

Target	Comment
source	Create a new source distribution of the . This target should normally only be executed by the editor of the . The name of the archive could e.g. be `nisp-src-3.0.zip` and will be placed in the subdirectory `zip`
tool	Create a new tool distribution of the . This target should normally only be executed by the maintainer of the nisp-tools package. The appropriate files from the tool directory are copied to the `dist` directory, and a new archive is created. The name of the archive could e.g. be named e.g `nisp-tool-3.1.zip` and will be placed in the subdirectory `zip`.
web	Create a archive of the files, necessary for a Web version of the . The name of the archive could e.g. be named e.g `nisp-web-3.0.zip` and will be placed in the subdirectory `zip`.
pdf-only	Create an archive with PDF versions of all the documents. The name of the archive could e.g. be named e.g `nisp-pdf-3.0.zip` and will be placed in the subdirectory `zip`.
xsl	Create a *stylesheet only* distribution. Used to release stylesheet distributions in between normal tools releases.
zip	Create the source, tool, xsl, web and pdf-only archives

7.3.4.2. Validating a document

It was previous mentioned, that there are Ant targets, which are used to validate XML document (i.e. that obey the syntax of the DocBook DTD). This is necessary, since existing many editors currently are not XML-aware, and it is therefore necessary for the author to ensure, that he is writing a syntactically correct DocBook document. Because the the DocBook syntax is very large, the user can not manually verify the correctness of the document, but must use a validating parser. Saxon comes with a non-validating parser, which is why the validating parser Xerces is included in the tools distribution and is used for this purpose.

When the user wants to validate volume 5 (under Windows), the following command is executed

```
.\build.bat valid5
```

This result in the following output from the validation process:

```
Buildfile: build.xml

valid5:
/home/js/work/noswg/xml-nisp/tools/src/volume5/vol5.xml: 3917 ms
        (3725 elems, 522 attrs, 10080 spaces, 133734 chars)

BUILD SUCCESSFUL

Total time: 6 seconds
```

The program used to validate a XML files, is called SAXCount and is a sample application included with the Xerces distribution. It is actually used to count the number of elements, attributes, text characters and ignorable white-space characters in the document. By default, we have turned on the validation flag, and all though the output does not say it, volume 5 is a valid document. If that was not the case, a large number of error messages describing the syntax errors, would be written to the standard output.

If the editor *jEdit* is used, this step is unnecessary. Whenever a document is opened, the focus shifts to a new document or a document is saved, the current buffer is automatically validated .

7.3.5. Building a internet version of the documents

In order to publish a version of the for the internet

7.4. Publishing the NISP

This section describes the steeps required, when releasing new distributions, and should normally only be followed by the editor and maintainer of the tools and stylesheet distribution.

Cleaning Up before a Build

Before releasing a new distribution, it is always a good idea, to run the target *clean*, which deletes the `build` and `dist` directories. During development, it is very likely, that these directory are filled up with uneeded file, when the user was experimenting with the build tools. So it is better to build everything from scratch, to ensure, that everything id created properly.

7.4.1. Preparing a Source Release

During the current lifecycle of the , besides the official releases in December, it is often convenient for the editor, to be able to release during development of a new version of the , to ensure that everybody uses the same baseline. The following steps should be taken by the editor, to ensure to release a new version of the sources.

1. *Cleanup the source directories* - During the development of the sources, a lot of *tempory* stuff is usually created, and unless these temporary files should be keept, which would only make sence if it is not an official release, A detailed inspection of all the source directories should be conducted, since everything except, what is explicitly stated in the `build.xml` configuration file from the tools distribution, will be included in the new source distribution.

2. *Update the source distribution version numbers* - Edit the `src/VERSION` file, which contains all version numbers for the architecture document, and the additional documents like the rationale document and the traceability matrix. Normally the entity variable `src-version-minor` should just be a number, typical 0 (zero), but in between official releases, this could be `0-pre1` or something similar. The editor should also update the entity variable `src-release-date`.

3. *Update source distribution Version History* - Modify the file `src/WhatsNew` with the information considered appropiate.

4. *Update source document numbers* - If we are dealing with an official, release. It is a good idea to make sure all documents contains the correct version numbers, and date. For the archtecture documents this can be accomplished by making sure that the first revision element is the following:

```
<revision>
  <revnumber>&src-version;</revnumber>
  <date>&src-release-date</date>
</revision>
```

The entity variables is from the just modified `src/VERSION`.

5. The last step is just to create a source distribution, by running the target *source* like already described in Section 7.3.2, “Running Ant from a Command Shell” and Section 7.3.3, “Running Ant from jEdit”. This step will create a compressed *zip* file, which will be alled e.g. `nisp-src-3.0.zip`, and which will be placed in the directory `zip`, which is located in the root of the nisp tools distribution.

7.4.2. Preparing an official release of the NISP

After a new source distribution have been created, with all the cleaning up, the task of creating a new release is very easy[4]. Just run the target *web*, which will create HTML and PDF versions of all documents, and also create the *homepage* for a web portal to the , and additional material. After the creation of the components, everything will be packed in a compressed *zip* file called e.g. `nisp-web-3.0.zip`, and which will be placed on the directory `zip`, whic is located in the root of the nisp tools distribution.

Some people would prefere a PDF only distribution. This is accomplished by running the target *pdf-only*. The will create a compressed *zip* file called e.g. `nisp-pdf-3.0.zip`, and which will be placed in the directory `zip`, which is located in the root of the nisp tools distribution.

7.4.3. Preparing a new Release of the Tools

7.4.4. Preparing an interrim Stylesheet Release

Since the DocBook XSL stylesheets are released quite frequently, it very possible that the need to release part of the tools distribution arises. Because the tools distribution is quite big, it is not convenient to release such a large distribution often.

This is the reason the stylesheet-only distribution was created. Whenever a new tools distribution is released e.g. *nisp-tools-3.1.zip*, we will also create a stylesheets distribution called e.g. *nisp-xsl-3.1.0-1.70.1.zip*, where the number *1.70.1* is the number of the latest DocBook XSL stylesheet distribution. N.B. this stylesheet distribution will allready be included in the tools distribution, and is only created if we want to return

[4]Remenber to first run the target *clean*, as mentioned in the sidebar.

to the original stylesheets. The name of the next stylesheet-only distribution could then be *nisp-xsl-3.1.1-X.Y.Z.zip*, where X.Y.Z is the number of the new DocBook XSL release, which might just as well be 1.70.1 as the previous version, if we want to fix bugs in our customizations, or just experiment with new designs.

8. Building the NISP

8.1. Introduction

The NISP source package contains all DocBook XML files which are used to transform to other formats. Besides the XML files, the sources also contains Scalable Vector Graphics (SVG) files, which is a also a XML format for 2D vector graphics. In the root of the source distribution, the file documents.xml contains a description of all documents, which are part of the NISP The data configuration data in the file is used to create the rules necessary to build HTML/PDF and a Windows HTMLHelp file of the NISP. This file is also used by some of the stylesheets, to extract document specific parameters.

The NISP tool package is used to transform the DocBook XML documents to XHTML, PDF and the Windows HTMLHelp format.

The tools used for this transformation is

- Xerces for parsing XML documents
- Saxon for transforming to XHTML and XSL-FO
- Fop for transforming XSL-FO to PDF
- Batik for transforming SVG to JPEG
- iText for post processing of the PDF file
- resolver for resolving public identifiers in DOCTYPE declarations

In order to automate the steps required to compile the NISP sources, the package Ant is used. Ant is a make-like tool implemented in Java, which can build targets defined in a configuration file called a build file. In the NISP ant is called from the script build.bat on Windows or build.sh on unix, which takes of declaring environment variables and loading of the required Java classes. To build a target for the NISP, the user should run the command:

```
build [-f buildfile] [target [target ...]]
```

If a build file is not specified the system will by default use the buildfile `build.xml`. If a target is not specified, the default target is used, for more information on available targets in a build file, run the command:

```
build [-f buildfile]  -projecthelp
```

Besides listing public targets, the command also shows the name of the default target, which will be used if no target have been specified.

In the NISP build system the two main build files are:

build.xml	This file contains the targets used to create the normal NATO/EAPC UNCLASSIFIED version
build-internet.xml	This file creates an unlabelled version of the NISP

The file `build.xml` contains all the target used to build the HTML/PDF and Windows HTML Help files and also target used to create archives of the website version, new tools and source packages and also targets used to clean-up the build and distributions directories. This file will be described in detail in Section 8.3, “Build targets”

The file `build-internet.xml` contains all the targets to build an unlabelled version of the technical architecture. The build file imports the build file `build.xml`, but provides only three targets web, pdf-only and clean.

8.2. Building a document

To transform a specific volume (e.g. volume 4), the user should run the command

```
build vol4
```

Running this target will automatically run a number of sub targets, which will essentially do the following:

- Create a resolved version, if the XML sources have been changed.
- Create JPG version of the SVG vector images, if any SVG image have been changed
- Create a XHTML version of the resolved version have been changed
- Create a PDF version if the resolved version or any of the SVG images have been changed

The term changed should be interpreted as "modified or if the resulting file/files does not exists"

8.2.1. Resolving a documents

Creating a resolved version of a given document is NISP specific step in the building process, and is actually used for a number of things.

- Add the two attributes id and condition to the book element. Both variables is extracted from the file `src/documents.xml` and a used to define a unique identifier for the book element and to define a subdirectory to the build directory, where the XHTML versions of the document should be stored.
- volume 3, 4, 5 and the rationale document should be merged with a database before being transformed into a another format. This step is accomplished by applying one of the stylesheets `merge3.xsl`, `merge4.xsl`, `merge5.xsl` and `merge-rd.xsl` to the source xml file.
- In the source documents all references to images are to a SVG file located in a subdirectory of the source document. In the build directory XHTML files should refer to JPG version of the image, and XSL-FO (the transformation step before creating the PDF file), should either refer to the JPG file used for the XHTML version or eventually to the original SVG image.

The manipulation of image file references mentioned above, the figure element

```
<figure id="v4f-spsa-co-osi">
  <title>Protocol Layers Used in a Typical OSI Communications Lin
  <mediaobject>
    <imageobject>
      <imagedata fileref="figures/v4-osi.svg"/>
    </imageobject>
  </mediaobject>
</figure>
```

is transformed to

```
<figure id="v4f-spsa-co-osi" float="0">
  <title>Protocol Layers Used in a Typical OSI Communications Link
  <mediaobject>
    <imageobject role="html">
      <imagedata fileref="figures/v4-osi.jpg"/>
    </imageobject>
    <imageobject role="fop">
```

```
        <imagedata fileref="../volume4/figures/v4-osi.jpg"/>
      </imageobject>
    </mediaobject>
  </figure>
```

When the actually transformation takes place, a stylesheet will select the appropriate imageobject element.

8.3. Build targets

The build file `build.xml` defines a number of targets used to generate XHTML/ PDF and a Windows HTML Help file of the different volumes, all of which depends of the number of documents in the NISP collection. It also contain a number of other more static targets used for different purposes.

8.3.1. Document targets

For each of the volumes in the technical architecture, there exists a number of target necessary to implement the algorithm described in Section 8.2, “Building a document”. Of the 11 targets currently defined only three are significant for the user, the rest of the target are intermediate targets. For volume 1 the targets are:

- vol1.html - build volume 1 in XHTML
- vol1.pdf - build volume 1 in PDF
- vol1 - build volume 1 in XHTML and PDF

There exists similar target for volume 2-5. For the rationale document the respective targets are called rd.html, rd.pdf and rd. Besides these target, a couple of generic targets also exists like html and pdf, which will create XHTML and PDF versions of all documents.

All the target for the different volumes are dynamically generated as described in Section 8.4, “Document configuration”.

In the root distribution, a special build directory called `build/` is created, where all the transformed document will be placed

8.3.2. Administrative targets

Among the targets used for administration are the following:

8.3.2.1. Building NISP distributions

web	build a zip file containg a complete set of XHTML/PDF and Windows HTML Help file
pdf-only	build a zip file containing only a PDF version of the technical architecture.

8.3.2.2. Builing Tool and source packages

tool	build a zip file with a new tool package
source	build a zip file with a new source distribution
xsl	build a zip file with only the stylesheets

8.3.2.3. Cleaning the distribution

clean-build	Erase the `build/` directory
clean-dist	Erase the `dist/` directory (used to build a new tool distribution)
clean	Run both clean-build and clean-dist

8.3.2.4. Other targets

8.4. Document configuration

The number of target for the individual volumes defined in the build file may vary over time, we may add or delete documents, and we will therefore also have to change the build instructions required. These instructions used to be written manually, but since that process very easy introduces errors in the building process.

All the targets used to build the different volumes of the technical architecture are actually generated by a different Ant configuration file, and the resulting rules are saved in the file `build.targets`, which will be included in the `build.xml` file as an external entity. If new documents are included in the distribution, the user should cre-

ate the appropriate directory in the source distribution, modify the file `src/documents.xml` and finally run the command

```
build -f newbuild.xml
```

A typical fragment, which describes the a specific volume is shown below:

```
<directory dir="volume4">
  <docinfo id="vol4">
    <main>vol4.xml</main>
    <figures/>
    <titles>
      <short>V4</short>
      <title>Volume 4</title>
      <longtitle>NC3 Common Standards Profile (NCSP)</longtitle>
    </titles>

    <resolve usedb="yes" db="ta-standards.xml">merge4.xsl</resolve

    <targets>
      <target type="html">index.html</target>
      <target type="pdf">NISP-Vol4</target>
    </targets>
  </docinfo>
</directory>
```

Most of the elements and attributes are selfexplanatoty, so we will only mention a couple of them:

TBD

Bibliography

Normative References

[UCS] .

[XML 1.0] Tim Bray. *Extensible Markup Language (XML) 1.0*. 2nd ed. W3C Recommendation. 6 October 2000. Copyright © 2000 The World Wide Web Consortium. http://www.w3.org/TR/2000/REC-xml-20001006/ .

[Names] Tim Bray. *Namespaces in XML*. W3C Recommendation. 14 January 1999. Copyright © 1999 The World Wide Web Consortium. http://www.w3.org/TR/1999/REC-xml-names-19990114/ .

[SVG] Jon Ferraiolo. *Scalable Vector Graphics (SVG) 1.0 Specification*. W3C Recommendation. 4 September 2001. Copyright © 2001 The World Wide Web Consortium. http://www.w3.org/TR/2001/REC-SVG-20010904/ .

[XInclude] Jonathan Microsoft March and David BEA Systems Orchard. *XML Inclusions (XInclude) 1.0*. W3C Candidate Recommendation. 17 September 2002. http://www.w3.org/TR/2002/CR-xinclude-20020917/ .

[Wa02a] Norman Walsh. *XML Catalogs*. Committee Specification. 6 August 2001. Copyright © 2000, 2001 The Organization for the Advancement of Structured Information Standards [OASIS]. http://www.oasis-open.org/committee/entity/spec.html .

[Wa02b] Norman Walsh. *The DocBook Document Type*. Committee Specification 4.2. 16 July 2002. Copyright © 2001, 2002 The Organization for the Advancement of Structured Information Standards [OASIS]. http://www.oasis-open.org/docbook/specs/cs-docbook-docbook-4.2.html .

[XSLT] James Clark. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation. 16 November 1999. Copyright © 1999 The World Wide Web Consortium. http://www.w3.org/TR/1999/REC-xslt-19991116/ .

[XPath] James Clark and Steve Derose. *XML Path Language (XPath) 1.0*. W3C Working Draft. 16 November 1999. Copyright © 1999 The World Wide Web Consortium. http://www.w3.org/TR/xpath/ .

[XSLT11] James Clark. *XSL Transformations (XSLT) Version 1.1*. W3C Working Draft. 24 August 2001. Copyright © 2001 The World Wide Web Consortium. http://www.w3.org/TR/xslt11/ .

[XSLT2] Michael Kay. *XSL Transformations (XSLT) Version 2.0*. W3C Working Draft. 16 August 2002. Copyright © 2002 The World Wide Web Consortium.

http://www.w3.org/TR/2002/xslt20/ .

[XSL] Sharon Adler and Steve Ziles. *Extensible Styesheet Language (XSL) Version 1.0*. W3C Recommendation. 15 October 2001. Copyright © 2001 The World Wide Web Consortium. http://www.w3.org/TR/xsl/ .

Articles

[Fl01] Peter Flynn. *The XML FAQ*. 2.1. 1 January 2002. http://www.ucc.ie/xml/faq.html .

[Ho00] G. Ken Holman. *What is XSLT?*. 16 Aug 2000. Copyright © 2000 xml.com. http://www.xml.com/pub/a/2000/08/holman/index.html .

[Ho02] G. Ken Holman. *What is XSL-FO?*. 20 Mar 2002. Copyright © 2002 xml.com. http://www.xml.com/pub/a/2002/03/20/xsl-fo.html .

[St02a] Robert Stayton. *A new implementation of Olink, Ver. 2.0*. Jul 14, 2002. Copyright © 2002 Robert Stayton. http://sagehill.net/xml/OlinkExtended.html/ .

[St03] Robert Stayton. *DocBook XSL : The Complete Guide*. 2.0. February 2005. Copyright © 2002, 2003, 2005 Robert Stayton. http://sagehill.net/docbokxsl/ .

[Wa98] Norman Walsh. *A Technical Introduction to XML?*. 3 Oct 1998. Copyright © 1998 xml.com. http://www.xml.com/pub/a/98/10/guide0.html .

[Wa00] Norman Walsh. *If you can Name it, you ca claim it!*. 4 Apr 2000. Copyright © 2000 Abortext, Inc.. http://www.abortext.com/Think_Thank/XML_Resources/Issue_Three/body_issue_three.html .

[Wa01a] Norman Walsh. *The Design of The DocBook XSL Stylesheets*. 8 Apr 2001. Copyright © 2001 Sun Microsystems Inc.. http://nwalsh.com/docs/articles/dbdesign/ .

[Wa01b] Norman Walsh. *XML Basics*. Copyright © 2001 Linux Magazine. http://www.linux-mag.com/2001-07/xml_basics_01.html .

Books

[Mu02] Chuck Musciano. *HTML & XHTML: The Definitive Guide*. 4th ed.. Copyright © 1996, 1997, 1998, 2000 O'Reilly & Assocates, Inc.. ISBN 0-596-00026-X.

[Ca01] Kurt Cagle, Michael Corning, Jason Diamont, and Teun Deynstee. *Professional XSL*. Copyright © 2001 Wrox Press Ltd.. ISBN 1-861003-57-9.

[HaMe01] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell - A Desktop Quick Reference*. Copyright © 2001 O'Reilly & Assocates, Inc.. ISBN

0-596-00058-8.

[Kay01] Michael Kay. *XSLT - Programmers Reference*. 2 ed. Copyright © 2001 Wrox Press Ltd.. ISBN 1-861005-06-7.

[Li99] Håkon Lie and Bert Boss. *Cascading Stylesheets*. Copyright © 1997, 1999 Pearson Education Limited. ISBN 0-201-59625-3.

[Ma01] Didier Martin, Michael Kay, and etc.. *Professional XML*. Copyright © 2000 Wrox Press Ltd.. ISBN 1-861003-11-0.

[Pa02] Dave Pawson. *XSL-FO*. Making XML Look Good in Print. Copyright © August 2002 O'Reilly & Assocates, Inc.. ISBN 0-596-00355-2.

[WaMu99] Norman Walsh and Leonard Muellner. *DocBook - The Definitive Guide 1.0.3*. Nov 9, 1999. Copyright © 1999 O'Reilly & Assocates, Inc.. ISBN 1-56592-580-7. http://docbook.org/tdg/ .

[WaMu05] Norman Walsh and Leonard Muellner. *DocBook - The Definitive Guide 2.0.12*. Apr 18, 2005. Copyright © 1999, 2000, 2001, 2002, 2003, 2004, 2005 O'Reilly & Assocates, Inc.. ISBN 1-56592-580-7. http://docbook.org/tdg/ .

A. Inside the NISP tools distribution

This appendix contains references to all the software packages used in the . The versions mentioned here, may not be the lastest versions, but have on the other hand been tested with the documents.

A.1. Directory Map

The following figure illustrates the structure of the NISP tools distribution.

```
-|--bin                - Script files
 |--extra              - additional material
 |--docs               - Documentation
 |--lib                - Contain jars for Ant, Xerces, Saxon, Fop
 |--schema             - Misc. extra schemas used by the tool pac
 |--src                - Empty directory, where sources should be
 |--xdoc               - Documentation sources
 |--xsl                - XSL stylesheets
     |--common         - Common stylesheets
     |--css            - Cascaded stylesheet used by HTML version
     |--docbook-xsl    - DocBook XSL stylesheets
     |   |--
     |   ...
     |
     |--fo             - NISP XSL stylesheets for FO
     |--html           - NISP XSL stylesheets for HTML
     |--htmlhelp       - NISP XSL stylesheets for Windows Help fi
     |--images         - Raster images used in the HTML version
     |--xhtml          - NISP XSL stylesheets for HTML
```

A.2. Overview

This section describes the necessary software packages, and describes how to install them. In order to simplify things, most of these tools have been bundled in the NISP Tools distribution.

Only part of nine of the packages comes pre-installed with the documents. The pre-installed packages are:

- The DocBook XML DTD

- The DocBook XSLT stylesheet
- The Apache XML-Commons library
- The XML Catalog Resolver library
- The Xerces XML parser
- The Saxon XSLT processor
- The Fop XSL-FO processor[1]
- The Batik svg processor
- The Ant build tool

A.2.1. Patching

Both the Saxon XSLT processor and the FOP XSL-FO processor have been patched and subsequently recpompiled to obtain functionality originally not availiable in the official distribution.

A.3. Software Requirements in the publishing cycle

Table A.1, “Software Required” illustrates the software packages necessary to work with the XML version of the NC3 . Each row list a software package, and each column list the functionality required by the user. The column headings should be interpreted as follows:

Edit	The packages required to edit the NISP XML version.
Validate	The packages required to validate if the documents follows the required syntax, ad defined in the DocBook XML DTD
HTML	The packages required to create a HTML version of the NISP
Print	The packages required to create a *print* version of the NISP. The result is a formatting object version, which then are transformed into PDF.

[1]This version have been pacthed to obtain specific functionality.

	Edit	Valid-ate	HTML	Print	
NISP sources	Yes	Yes	Yes	Yes	
NISP Tools		Yes	Yes	Yes	
Java Runtime Environment 1.4.2_06	(Yes)	Yes	Yes	Yes	
Resolver 1.1	(Yes)	Yes	Yes	Yes	Included in the tool distribution
DocBook XML DTD 4.3		Yes	Yes	Yes	Included in the tool distribution
Xerces Parser 2.9.0		Yes			Included in the tool distribution
XSL Stylesheets 1.70.1			Yes	Yes	Included in the tool distribution
Saxon 6.5.5			Yes	Yes	Included in the tool distribution
FOP 0.20.5			Yes	Yes	Included in the tool distribution
Batik 1.6			Yes	Yes	Included in the tool distribution
Ant 1.6.5		(Yes)	(Yes)	(Yes)	Included in the tool distribution

Table A.1. Software Required

A.4. Release Information

The NISP Tools

File	:	nisp-tools-3.1.zip
Released	:	Sep 17, 2009
Comment	:	This is the tools used to build a HTML and print versions of the NATO C3 . See the README file for a detailed description for the contents of this archive

The NISP sources

File	:	nisp-src-3.0.zip
Released	:	9 February 2009

Comment	:	This is the NATO C3 implemented in XML. See the README file for a detailed description for the contents of this archive

DocBook XML DTD 4.3

Released	:	April 1, 2004
Comment	:	This is the XML version of the DocBook DTD
Homepage	:	http://www.oasis-open.org/docbook/

Java Runtime Environment 1.4.2_06

Released	:	May 2001
Comment	:	One can also download the Java 2 Software Development Kit, but that is currently not necessary
Homepage	:	http://java.sun.com/j2se/1.3/jre/

Resolver 1.1

Released	:	Jan 9, 2002
Comment	:	The resolver classes are used to map public and system identifiers and URIs to file names or other URIs.
Homepage	:	http://www.sun.com/software/xml/developer/resolver/

Xerces Parser 2.9.0

Released	:	November 20, 2003
Comment	:	This is a validating XML parser
Homepage	:	http://xml.apache.org/xerces-j/

XSL Stylesheets 1.70.1

Released	:	Aug 12, 2005
Comment	:	XSLT stylesheets used to generate the HTML and PDF version
Homepage	:	http://docbook.sourceforge.net/

Saxon 6.5.5

Released	:	Nov 24, 2005
Comment	:	This is the XSLT processor used to transform the XML sources to HTML and PDF
Homepage	:	http://saxon.sourceforge.net/

Batik 1.6

Released	:	Apr 11, 2005
Comment	:	This is the SVG processer, which generate raster images of all the figures
Homepage	:	http://xml.apache.org/batik/

FOP 0.20.5

Released	:	Feb 20, 2004
Comment	:	This is the XSL-FO processer, which generate PDF versions of the . The version used have been modified to enable setting the Creation Date of a document in the Document properties dialog.
Homepage	:	http://xml.apache.org/fop/

Ant 1.6.5

Released	:	Jun 2, 2005
Comment	:	A make like system from the Apache Project. This make like system is used to run all the commands necessary to build all documents both in a HTML and a print version of the .
Homepage	:	http://ant.apache.org/

A.5. Building the TA

B. TO DO

This chapter describes a number of potential improvements to the tools package, that might be implemented in a future version. This is only a catalog of ideas, and should in now way be considered as something which will be implemented really soon now.

- *Tracking* - It would be nice, if it was possible to create printable versions, where the difference between two versions of a document could be shown. What is added, What is deleted etc.

- *Index* - In DocBook it is possible to create an index[1]. However such an index requires planning. This is also almost a source-only distribution problem.

- *Modular Documents* - Currently most documents in the consists of one large file. It is currently possible to to split a document into manageable portions, but only through external entities. This means, that the sub documents, are not allowed to contain any XML prologue statement, or any document type definition, which in the end means. that these document fragments can not be validated. The candidate recommendation XML Include (see [XInclude]) makes a properly modular structure possible, where each document fragment is also a valid XML document. However this is not very well supported yet, at least not by the Xerces parser which is used to parse the documents.

- *Using Resolution Dependent Images* - The images are currently transformed to a raster for suitable for Screen (i.e. 70 dpi). The images are also used for the FO version. The ideal solution would be to create an extra collection of images for the FO version with and approximate resolution of 110 dpi.

- *Embedding SVG images* - With a collection of images all created in SVG. The ideal solution, would be use a mixed namespace solution and embed the images in both the Web Browser and in a FO representation of the document. But until Web browsers comes with native support for SVG and not through a third party plugin like the *Adobe SVG Viewer*, that is not an option at least for the HTML solution. For the FO version, the situation is a little different, since one of the main developers of FOP are also involved in the Batik project, and therefor also working with techniques for seamless integration of these two products. To implement part of this in the technical architecture, experiments need not be conducted to judge, if the results will be as excellent as the current raster images.

- *Using Portable Web Fonts* - The current SVG standard is quite vague, when it comes to the use of fonts in SVG images. Most of the components of a SVG image is independent of the platform and therefore extremely portable. The only problem is in the use of fonts. As long as the standard does not define and include a minimum recommended set of fonts, the creator of SVG images are depend either on the set of fonts available on the current platform, or included on the application used to create the SVG image. For the , this currently means, that the images, which are

[1] A standard index was created for vol 3 and 4 in version 4.4

created on a Windows platform needs to be rendered on the same platform, otherwise the Batik processor with substitute the Windows only fonts, with similar types on different platforms with undesired results, since the glyphs of different fonts will have a different design. It will therefore also be a problem, if some SVG images are created on a machine with an exotic but not standard font, which often is the case when receiving document, presentations etc. from miscellaneous parties.

- *Spell Checking* - Spell checking of the documents, should be included by default. The jEdit editor already have support for spellchecking and is working like a snap on Unix. Support for Windows do exists, though the open-source tool aspell. Experiments need to be done, to investiate how this is done without any hazzle.

- *Configuration Control* - With multiple authors working on documents in parallel, a configuration control system would be very convenient. On most large open source development systems, this is mandatory. According to one of the well-known development sites (SourceForge) : *Sourceforge.net is the worlds largest development website with the largest repository of Open Source code and applications avalible on the internet.* This of course requires strict configuration control and on this site are examples on tools, that enables such an development environment, even on the NISP sources, through the use of encryption software. See among other the manual on using CVS and SSH on the Windows platform: https://sourceforge.net/docman/display_doc.php?docid=766&group_id=1. Implementing such a system would require all users to be able to access a CVS server from the normal environment. This might not be possible due to security restrictions forced upon them,