BRICS

Basic Research in Computer Science

The XML Revolution

Anders Møller
Michael I. Schwartzbach

BRICS Notes Series **NS-00-8**

ISSN 0909-3206 **December 2000**

See back inner page for a list of recent BRICS Notes Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/00/8/`

The XML Revolution

Technologies for the future Web

Anders Møller &
Michael I. Schwartzbach

`http://www.brics.dk/~amoeller/XML/`

This 130+ page slide collection provides an introduction and overview of **XML**, **Namespaces**, **XLink**, **XPointer**, **XPath**, **DSD**, **XSLT**, and **XML-QL**, including selected links to more information about each topic.

About these pages...

This slide collection about XML and related technologies is created by

Anders Møller
http://www.brics.dk/~amoeller

and

Michael I. Schwartzbach
http://www.brics.dk/~mis

for use in an XML course at University of Aarhus, Denmark.

The slide collection is primarily aimed at computer scientists, software engineers, and others who want to know what this XML thing is all about. It covers both the basic XML concepts and the related technologies for document linking, describing classes of documents, stylesheet transformation, and database-like querying. The slides are designed with concrete motivation and technical contents in focus, for the reader who wishes to understand and actually use these technologies.

The slides were last updated *June 2000*.

Feedback is appreciated! Please send comments and suggestions to amoeller@brics.dk.

Contents

XML - eXtensible Markup Language

Why XML?

The Web today: everything is **HTML** - *HyperText Markup Language*.

- **Hypertext:** text with links to other texts.
- **Markup Language:** annotations for structure of a text.

HTML design:

- HTML markup is designed to provide **logical structure** of information intended for presentation as **Web pages**.
- HTML contains a **fixed set of markup tags**, and also defines their semantics.

Example (from this page):

```
<h1>Why XML?</h1>
The Web today: everything is <b><a href="http://www.w3.org/MarkUp/">HTML</a></b> -
<i>HyperText Markup Language</i>.
```

HTML syntax is formally defined (but most HTML is not valid).

CSS (Cascading Style Sheets) allow separation of logical structure and **page layout**.

But: Although exact layout can be defined separately in CSS, HTML is designed for hypertext - *not for information in general!*

A brief history of HTML

HTML was designed in 1992 by Tim Berners-Lee and Robert Caillau at CERN.

HTML is an instance of SGML, Standard Generalized Markup Language.

Originally, HTML specified strictly the logical structure, not the physical layout.

Pressure from users (mainly industry), forced subsequent version to allow increasingly fine-grained control of the appearance:

1992

HTML is first defined

1993

HTML+ (some physical layout, fill-out forms, tables, math)

1994

HTML 2.0 (standard for core features)
HTML 3.0 (an extension of HTML+ submitted as a draft standard)

1995

Netscape-specific non-standard HTML appears

1996

Competing Netscape and Explorer versions of HTML
HTML 3.2 (standard based on current practices)

1997

HTML 4.0 (separates structure and presentation with style sheets)

1999

HTML 4.01 (slight modifications only)

2000

XHTML 1.0 (XML version of HTML 4.01)

Example: recipe collection in HTML

```
<h1>Rhubarb Cobbler</h1>
<h2>Maggie.Herrick@bbs.mhv.net</h2>
<h3>Wed, 14 Jun 95</h3>

Rhubarb Cobbler made with bananas as the main sweetener.
It was delicious.  Basicly it was

  <table>
  <tr><td> 2 1/2 cups <td> diced rhubarb (blanched with boiling water, drain)
  <tr><td> 2 tablespoons <td> sugar
  <tr><td> 2 <td> fairly ripe bananas sliced 1/4" round
  <tr><td> 1/4 teaspoon <td> cinnamon
  <tr><td> dash of <td> nutmeg
  </table>

Combine all and use as cobbler, pie, or crisp.

Related recipes: <a href="#GardenQuiche">Garden Quiche</a>
```

The ideas behind XML

- we need something more SGML-like: general meta-information

The XML design

- separates **syntax** (structural representation) from **semantics** (visual rendering or other processing) - and only considers syntax
- contains **no fixed set of markup tags** - we may define our own tags, tailored for our kind of information
- has built-in **internationalization** (full Unicode) and **platform independence**

The vision: *XML as the universal format for structuring information* - no more proprietary/obscure/bit-hacking/inflexible/incompatible formats.

Rendering in browsers is completely defined by **style sheets** (e.g. CSS or XSL).

The basic benefit of XML: **Generic tools** for e.g. **querying** and **transformation** can be made.

Example: recipe collection in XML

```
<recipe id="117" category="dessert">
  <title>Rhubarb Cobbler</title>
  <author><email>Maggie.Herrick@bbs.mhv.net</email></author>
  <date>Wed, 14 Jun 95</date>

  <description>
    Rhubarb Cobbler made with bananas as the main sweetener. It was delicious.
  </description>

  <ingredients>
    ...
  </ingredients>

  <preparation>
    Combine all and use as cobbler, pie, or crisp.
  </preparation>

  <related url="#GardenQuiche">Garden Quiche</related>
</recipe>
```

Illustrates

- markup purely for logical structure
- just one choice of markup detail level
- need for a kind of "grammar" for XML recipe collections
- need for stylesheet to define presentation semantics

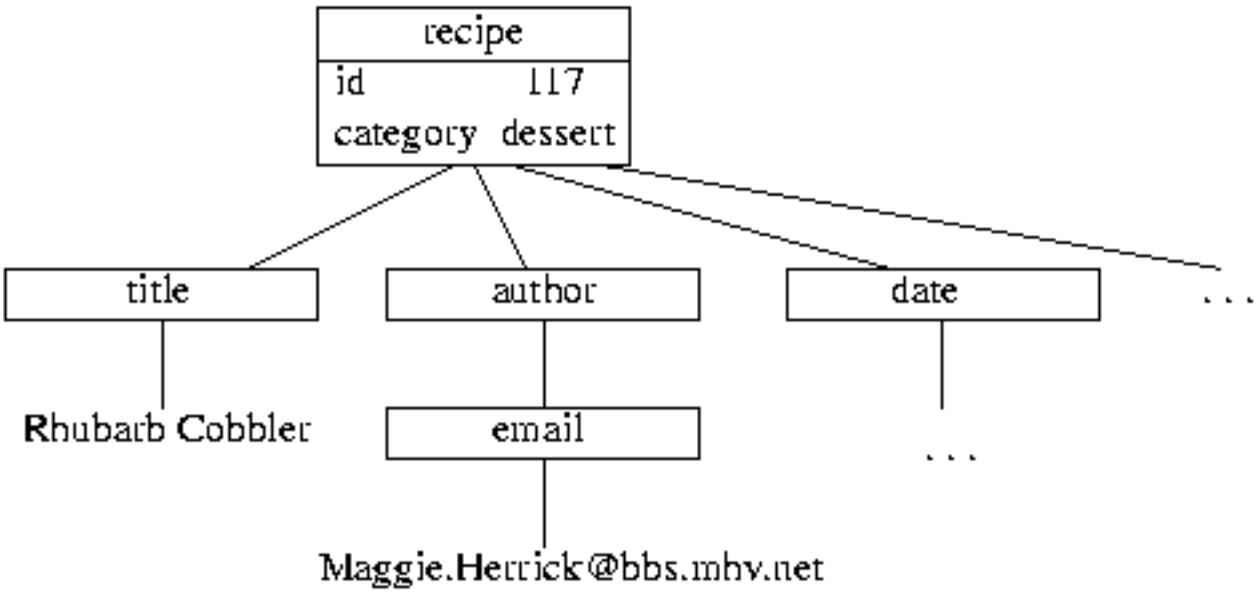
So what is it, really?

From a conceptual point of view

An XML document is a **labeled tree**.

- a leaf node is
  - **character data** (a text string) - the actual data,
  - a **processing instruction** - annotations for processors, typically in document header,
  - a **comment** - never any semantics attached,
  - an **entity declaration** - simple macros, or
  - DTD (Document Type Declaration) nodes (described later...)
- an internal node is an **element**, which is labeled with
  - a name, and
  - a set of **attributes**, each consisting of a name and a value.

Often, *comments*, *entity declarations*, and *DTD information* is not explicitly represented in the tree.



(Some prefer to have a special *root node* above the root element.)

Unfortunately, there is still no agreement on XML tree terminology :-(

So what is it, really?

From a concrete point of view

An XML document is a (Unicode) text with **markup tags** and other meta-information.

Markup tags denote elements:

```
...<bla attr="val" ...>...</bla>...
   |    |                 | |
   |    |                 | a matching element end tag
   |    |                 the contents of the element
   |    an attribute with name attr and value val, values enclosed by ' or "
   an element start tag with name bla
```

Short-hand notation for empty elements: `...<bla attr="val".../>...`

(Note: XML is *case sensitive!*)

Well-formed documents:

- start and end tags must match
- element tags must be properly nested
- (+ other syntactical requirements)

Special characters can be escaped using Unicode *character references*, example:

`&` = `&` `<` = `<` = `<`

CDATA Sections are an alternative to escaping in character data, example:

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

Note: Parsing is trivial! (in principle)

Other meta-information

`<!-- comment -->`

a comment, will be ignored by all processors

`<?target data...?>`

an instruction for a processor, `target` identifies the processor for which it is directed, `data` is a string containing the instruction

`<!ENTITY name value>`

declares an entity with a name and a value, expanded using *entity reference*: `&name;` (*external entities* and *parameter-entity references* are ignored here...)

`<!ELEMENT ...>`, `<!ATTLIST ...>`, ...

DTD information (briefly mentioned later... - better alternatives: DSD, XML Schema)

A larger example

- an article XML document

Illustrates

- snippets from a real-world XML application
- how to design an XML markup language
- relation to DSD and XSL (continued in later sections...)

Our notion of articles involves concepts such as

- titles, author lists,
- abstracts, sections, subsections, vitae sections, etc.
- cross references,
- citation references
- inline images, tables, lists, etc.

These concepts are *explicitly* present in the XML document as tailor-made markup -- it looks like HTML, but it's **purely logical structure**, no layout or presentation information. (However, it is designed with later presentation in mind.)

Later:

- **DSD** will later be used to define our class of article XML documents.
- **XSLT** will be used to transform the XML document into HTML, XHTML, or LaTeX, including automatic construction of index, references, etc.
- **XLink**, **XPointer**, and **XPath** will be used to create cross-references between articles.
- **XML-QL** will be used for queries on articles.

Example: `article.xml`, Part I

```
<?xml version="1.0" encoding="iso-8859-1"?>
<?dsd URI="article.dsd"?>
<?xml:stylesheet type="text/xsl" href="article2html.xsl"?>

<!--this document is available at http://www.brics.dk/DSD/examples/article.xml-->

<article postscript="http://www.brics.dk/DSD/papers/dsd.ps"
         pdf="http://www.brics.dk/DSD/papers/dsd.pdf">

<title>DSD: A Schema Language for XML</title>

<authors>
  <names>Nils Klarlund</names>
  <affiliation>AT&amp;T Labs Research</affiliation>
  <link href="mailto:klarlund@research.att.com">klarlund@research.att.com</link>
</authors>
<authors>
  <names>Anders Møller <amp/> Michael I.<nbsp/>Schwartzbach</names>
  <affiliation>BRICS, University of Aarhus</affiliation>
  <link href="mailto:amoeller@brics.dk,mis@brics.dk">{amoeller,mis}@brics.dk</link>
</authors>

<abstract>
We present DSD (Document Structure Description), which is a schema
language for XML documents. A DSD is itself an XML document, which
describes a family of XML application documents.
<p/>
...
</abstract>
```

Example: `article.xml`, Part II

```
<section id="introduction">
<title>Introduction</title>

A Document Structure Description (DSD) is a specification of a
class of XML documents. A DSD defines a grammar for XML documents,
default element attributes and content, and documentation of the
class. A DSD is itself an XML document.  We have five major goals for the
descriptive power of DSDs, namely that they should:

<itemize>
<item>
allow context dependent descriptions of content and attributes, since
the context of a node, such as ancestors and attribute values, often
govern what is legal syntax;
</item>
<item>
generalize CSS<nbsp/><cite id="bos98:_cascad_style_sheet_css2_specif"/>
(Cascading Style Sheets) so that readable, CSS-like rules for default attribute
values and default content can be defined for arbitrary XML domains, not only
predefined user formatting models;
...
</itemize>

This processing instruction has the form
<oneline>
<tt>&lt;?dsd URI="</tt><it>URI</it><tt>"?&gt;</tt>
</oneline>
```

Example: `article.xml`, Part III

```
...the book example described in Section<nbsp/><ref idref="sec:bookexample"/>.

These two kinds of <tt>title</tt> elements can be defined as follows:
<example><![CDATA[
<ElementDef ID="book-title" Name="title"
            Defaultable="yes">
  <Content><StringType/></Content>
</ElementDef>
]]></example>

<references>
<item id="dsddoc99">
  <authors>Nils Klarlund, Anders Møller, and Michael I. Schwartzbach</authors>
  <title>Document Structure Description 1.0</title>
  <publisher>AT<amp/>T <amp/> BRICS</publisher>
  <month>October</month><year>1999</year>
</item>
</references>

<vitae>
<person img="http://www.brics.dk/DSD/examples/klarlund.jpg"
        alt="klarlund@research.att.com" width="100">
<bf>Nils Klarlund</bf> received his Ph.D.<nbsp/>(Liberal Arts) from
Finkelstein Mail-Order College in 1989. He has bungled through life
since then, before remarkably landing a real job a AT<amp/>T, whose
stock value has subsequently plunged 43<percent/>. By the generosity of
numerous co-authors, his name appears on several publications.
<br/><br/>
<it>Homepage:</it> <link href="http://www.research.att.com/~klarlund/"
>http://www.research.att.com/<tilde/>klarlund/</link>
</person>
```

A note on DTDs

DTD: Document Type Definition

- a way of describing classes of XML documents (like grammars for other languages)

Serious problems with DTD:

- much too simple: not enough expressive power
- much too complex: horrible syntax (not even XML :-)
- mixed into the XML specification

A tiny example DTD:

```
 <!DOCTYPE recipecollection [
   ...
   <!ELEMENT recipe
(title,author?,date?,description,ingredients,preparation,related)>
   <!ATTLIST recipe id       ID                                      #REQUIRED
                    category (breakfast|lunch|dinner|dessert|unknown) #IMPLIED>
   <!ELEMENT title (#PCDATA)>
   <!ELEMENT author ANY>
   ...
 ]>
```

The solution: **schema languages** (DSD, XML Schema, ...)

For the record:
Well-formed documents: conform to common XML syntax rules
Valid documents: well-formed + conform to given DTD

Applications of XML

A few examples:

XHTML (www.w3.org/TR/xhtml1)

W3C's XMLization of HTML 4.0. Example XHTML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head><title>Hello world!</title></head>
  <body><p>foobar</p></body>
</html>
```

CML (www.xml-cml.org)

Chemical Markup Language. Example CML document snippet:

```
<molecule id="METHANOL">
  <atomArray>
    <stringArray builtin="elementType">C O H H H H</stringArray>
    <floatArray builtin="x3" units="pm">-0.748 0.558 -1.293 -1.263 -0.699
0.716</floatArray>
  </atomArray>
</molecule>
```

OMF

Weather Observation Markup Format - for weather observation reports.

XPML

Extensible Phone Markup Language - for interactive voice applications.

[www.oasis-open.org/cover/xml.html#applications contains a huge list of serious XML applications]

DOM - Document Object Model

- a platform- and language-neutral standard **API** (Applications Programming Interface) for manipulating XML (and HTML) **document trees**.

Ideal for scripting languages, e.g. ECMAScript.

Defined in Levels:

Level 0: existing functions known from browser scripting languages

Level 1: functionality for document navigation and manipulation (the "core")

Level 2: adds style sheet model, filters, event model, namespace support

Level 3: adds loading and saving, DTDs, schemas, document views and formatting

Status: W3C working at Level 2

SAX - Simple API for XML

- **event-based** (parsing events reported to application through callback).

Allows "lazy" construction of XML tree.

[more info: www.w3.org/DOM, www.megginson.com/SAX, developerlife.com/saxvsdom]

Tools

Parsers

- **Expat** (www.jclark.com/xml/expat.html)
  Written in C (ported to other languages), used by LIBWWW, Apache, Netscape, DSD, ...
- **XML4J** (www.alphaworks.ibm.com/tech/xml)
  From alphaWorks, in Java, based on Apache Xerces, supports DOM and SAX
- + 1000 others...

Editors

- **Xeena** (www.alphaWorks.ibm.com/tech/xeena)
  From alphaWorks, in Java, with tree-view syntax directed editing
- **XMLSpy** (www.xmlspy.com)
  Popular, but not free :-(
- + 1000 others...

Browsers

- **Netscape Navigator 6 and Internet Explorer 5**
  XML parsing and validation, rendering with XSL and CSS, script access via DOM
  IE is leading... :-(

  [More info: www.xmlsoftware.com has a comprehensive list of XML tools]

From SGML to SML

- DocHeads vs. Simpletons

SGML (Standard Generalized Markup Language)

ISO standard, 1985.
Huge amount of "document archive" applications in government, military, industry, academia, ...
A successfull well-known application: HTML is designed as a simple application of SGML.

|
V

XML

W3C Recommendation 1998.
A simple subset of SGML. Targeted for Web applications. Now **de facto standard**.

|
V

Canonical XML (www.w3.org/TR/xml-c14n)

W3C Working Draft, June 2000.
No DTD or general entity references, "canonical" representation.

SML (Simple Markup Language) (www.xml.com/pub/1999/11/sml >and www.xmlhack.com/read.php?item=205)

Web community discussions, 1999
Even simpler: no processing instructions or comments, UTF-8 and UTF-16 only, considerations on element attributes, white-space,...

Occam's razor: "*one should not increase, beyond what is necessary, the number of entities required to explain anything*"

Technologies building on top of XML

- a notation for trees is not enough...

The real force of XML is **generic languages and tools**!

We need:

namespaces

- to avoid name clashes when a document uses several "sub-languages"

schemas (grammars)

- to define classes of documents (DTD is not good enough...)

linking between documents

- a generalization of HTML anchors and links

addressing parts of documents

- it is not enough that only the author can place anchors

transformation

- conversion from one document class to another

querying

- extraction of information

Each of these issues will be described later.

Other related technologies (not covered here):

RDF (Resource Description Framework)

- a framework for *metadata* (statements about properties and relationships)

XML-Signature

- digital signatures of Web resources

XML Fragment Interchange

- dealing with fragments of XML documents

Conclusion

XML is

- hot ($$$),
- the standard for representation of Web information,
- supported by lots of generic tools,
- by itself, just a notation for hierarchically structured text.

Links to more information

www.w3.org/TR/REC-xml.html

the XML 1.0 specification

www.w3.org/XML

W3C's XML homepage

www.xml.com/axml/testaxml.htm

the Annotated XML Specification, by Tim Bray

pdbeam.uwaterloo.ca/~rlander

an online XML introduction and tutorial

www.xml.com

a great source of XML information

www.oasis-open.org/cover

another great source of XML information

metalab.unc.edu/xml

yet another great source of XML information

inf2.pira.co.uk/top011a.htm

one more great source of XML information

wdvl.internet.com/Authoring/Languages/XML

another great source of XML information

www.w3schools.com/xml

yet another great source of XML information

www.garshol.priv.no/download/xmltools

a good list of free XML tools

www.xmlhack.com

XML development news

news:comp.text.xml

XML newsgroup

www.ucc.ie/xml

FAQ

Namespaces

Motivation

- name clashes.

Consider an XML language WidgetML which uses XHTML as a sublanguage for help messages:

```
<widget type="gadget">
  <head size="medium"/>
  <big><subwidget ref="gizmo"/></big>
  <info>
    <head>
      <title>Description of gadget</title>
    </head>
    <body>
      <h1>Gadget</h1>
      A gadget contains a big gizmo
    </body>
  </info>
</widget>
```

Meaning of `head` and `big` depends on context!

This complicates things for processors and might even cause ambiguities.

The problem is: one common name-space.

Qualifying names

Simple solution: **qualify names with URIs** (Universal Resource Identifiers)

```
<{http://www.w3.org/TR/xhtml1}head>
  \                           / \  /
   --------------------------   --
       qualifying URI        local name
```

- same with attribute names.

Note: URI only used for identification - *doesn't have to point at anything*.

This is the idea... - but not *exactly* how it is done.

Namespace declarations

- indirection: namespaces are **declared** by special attributes and associated **prefixes**

Example:

```
<... xmlns:foo="http://www.w3.org/TR/xhtml1">
  ...
  <foo:head>...</foo:head>
  ...
</...>
```

`xmlns:`*`prefix`*`="`*`URI`*`"` declares a namespace with a prefix and a URI.

- Scope of declaration: **lexical** (the element containing the declaration and all descendants - can be overridden by nested declaration)
- Both element and attribute names can be qualified with namespaces.

Note: the prefix is just a proxy - applications should use only the URI.

The default namespace

- for backward compatibility and simplicity.

Unprefixed element names are assigned a **default** namespace.

- declaration: `xmlns="`*`URI`*`"`
- default value: "" (means: treat as unqualified name)
- does *not* affect unprefixed **attribute** names (they belong to the same namespace as their elements)

An example: WidgetML with namespaces

```
<widget xmlns="http://www.widget.org"
        xmlns:xhtml="http://www.w3.org/TR/xhtml1"
        type="gadget">
  <head size="medium"/>
  <big><subwidget ref="gizmo"/></big>
  <info>
    <xhtml:head>
      <xhtml:title>Description of gadget</xhtml:title>
    </xhtml:head>
    <xhtml:body>
      <xhtml:h1>Gadget</xhtml:h1>
      A gadget contains a big gizmo
    </xhtml:body>
  </info>
</widget>
```

Now the main part of WidgetML uses the default namespace which has the URI `http://www.widget.org`;
XHTML uses the namespace prefix `xhtml` which is assigned the URI `http://www.w3.org/TR/xhtml1`.

Consequences for other XML technologies

- namespace awareness.

XML languages and applications should consider Namespaces as an inherent part of XML.

parsers

- need an extra processing layer on top to resolve namespace declarations and uses

schemas

- schemas should control declarations and uses
- namespace URIs could be tied together with schemas

and so on...

Links to more information

www.w3.org/TR/REC-xml-names

The W3C XML Namespace Recommendation

www.jclark.com/xml/xmlns.htm

an explanation of the recommendation by James Clark

www.xml.com/xml/pub/1999/01/namespaces.html

an XML.com article on Namespaces

DSD - Document Structure Description

Grammars for XML documents

Schema languages

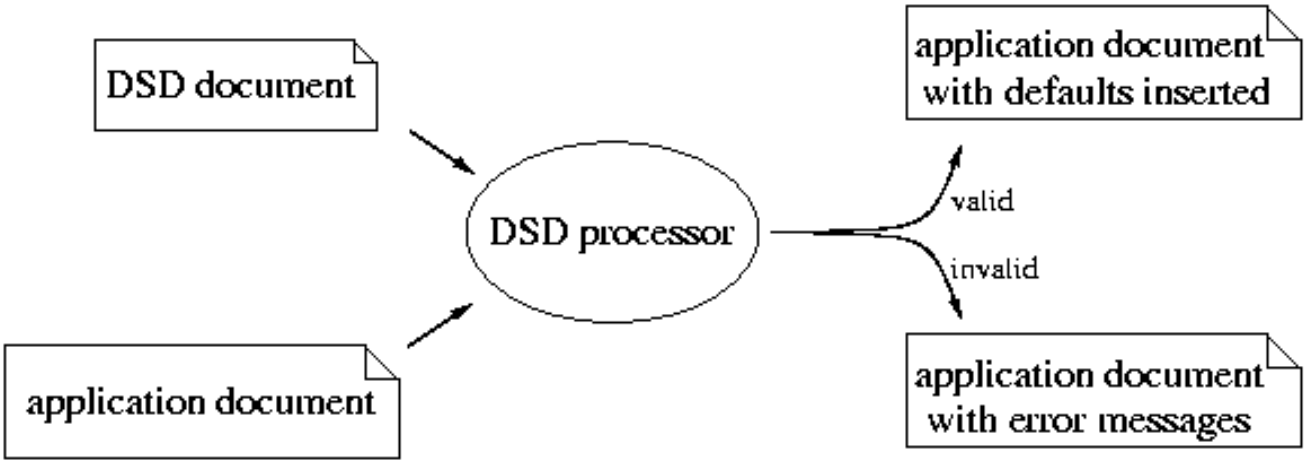
A schema defines the **syntax** of XML documents for a particular **application domain**.
(- in other words, it defines the grammar for an XML-based markup language.)

DTD (Document Type Definition): XML's native schema language

- not enough expressive power
- horrible syntax

DSD (Document Structure Description): AT&T and BRICS's schema language proposal

- very expressive
- is itself written in XML notation
- based on familiar computer science concepts (regular expressions, boolean logic, ...)
- 100% self-describable
- allows linear-time processing
- is defined in a short, human-readable specification



Using schema validators

A DSD description of a markup language provides a **formal** but **human-readable specification** of the syntax of the language.

A DSD processor can be generally useful both on the **server side** (when writing XML documents) and on the **client side** (when processing XML documents) on the Web:

- checking **validity** (conformance) of XML documents
- performing **default insertion** (inserts missing fragments)

Futhermore, a DSD processor can annotate XML documents with **parsing information** (IDs), useful for later processing.

Design requirements

- how should a schema language be designed?

From the W3C Note "*XML Schema Requirements*" (Feb. 1999):

Design principles: The XML schema language shall be

1. **more expressive than XML DTDs**;
2. **expressed in XML**;
3. **self-describing**;
4. usable by a wide variety of applications that employ XML;
5. straightforwardly usable on the Internet;
6. optimized for interoperability;
7. **simple** enough to be implemented with modest design and runtime resources;
8. **coordinated with relevant W3C specs**.

Structural requirements: The XML schema language must define:

1. mechanisms for **constraining document structure** (namespaces, elements, attributes) and **content** (datatypes, entities, notations);
2. mechanisms to enable **inheritance** for element, attribute, and datatype definitions;
3. mechanism for **URI reference to standard semantic understanding** of a construct;
4. mechanism for **embedded documentation**;
5. mechanism for application-specific constraints and descriptions;
6. mechanisms for addressing the **evolution** of schemata;
7. mechanisms to enable integration of structural schemas with **primitive data types**.

W3C's schema language, **XML Schema**, has been defined with these criteria in mind. In our opinion, XML Schema

- is not expressive enough,
- is not self-describing,
- definitely not simple,
- does not properly address schema evolution.

The DSD language provides an alternative, which (we believe) satisfies the requirements.

The DSD processing model

A DSD processor performs a single **top-down traversal** of the application document tree.

- During this traversal, each element node is assigned an **element ID**.
- An *element ID* determines an element **name** and a **constraint**.
- A *constraint* imposes requirements on the element **attributes** and **content**.

The DSD assigns a fixed element ID to the root element;
when checking the contents of an element, its sub-elements are assigned element IDs.

Terminology: the **current element** is the node currently being visited during the traversal.

The IDs function as **grammar nonterminals**.
(Other schema languages refer to e.g. element descriptions by *element name*, prohibiting multiple uses of the same element name.)

(Note: an element ID denotes a particular *kind* of element - the term "element ID" does not imply that two application document nodes cannot be assigned the same element ID.)

Components of a DSD

A DSD consists of a **header** and number of **structure definitions**, each associated with an ID for reference.

- `ElementDef` - an *element description*
  is a pair of an element name and a constraint.
- `ConstraintDef` - a *constraint*
  expresses requirements for an element's attributes, content, and context.
- `AttributeDeclDef` - an *attribute declaration*
  declares an attribute with a name and a string type.
- `ContentDef` - a *content expression*
  denotes a requirement for the content node-sequence.
- `BoolDef` - a *boolean expression*
  combines evaluation of attribute values and context.
- `ContextDef` - a *context expression*
  denotes a set of ancestor sequences.
- `StringTypeDef` - a *string type*
  is a regular expression denoting a set of character data strings or attribute values.

The *header* contains DSD meta-information: `Title`, `Version`, `Author`.

The DSD *root element* contains an attribute with a reference to the element ID assigned to the application document root element.

A tiny example

- a DSD for *business cards*, a class of XML documents

An application document `john_doe.xml`:

```
<card type="simple">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 456-1414</phone>
</card>
```

The main part of the DSD document `business_card.dsd`:

```
<DSD IDRef="card" DSDVersion="1.0">
  <Title> This is a DSD for XML business cards </Title>
  <ElementDef ID="card">
    <AttributeDecl Name="type" Optional="yes">
      <Union><String Value="simple"/><String Value="complex"/></Union>
    </AttributeDecl>
    <Element Name="name"><StringType/></Element>
    <If><Attribute Name="type" Value="simple"/>
      <Then> ... </Then>
    </If>
    ...
  </ElementDef>
  ...
</DSD>
```

Element descriptions

An element description associates a *name* and a *constraint* to an *element ID*.

```
elementdescr  ->  <ElementDef ID="element ID" Name="...">
                    constraintexp
                  </ElementDef>
```

If the `Name` is omitted, the `ID` is used as name.

(Note: some details are omitted in the grammars on the slides - see the specification.)

Constraints

A constraint can declare attributes, contains conditional subconstraints, and constrain element contents, attributes, and context.

```
constraintexp  ->  constraintterm*

constraintterm  ->  attributedecl |
                    contentexp |
                    boolexp |
                    constraint |
                    <If> boolexp
                      <Then> constraintexp </Then>
                      (<Else> constraintexp </Else>)?
                    </If>
```

Constraints are **evaluated** wrt. the **current element**.

Reusing expressions

Expressions can be reused using **definitions** and **references**:

```
constraintdef  ->  <ConstraintDef ID="...">
                     constraintexp
                   </ConstraintDef>

constraint  ->  <Constraint IDRef="..."/>
```

- same for other syntactic categories!

Boolean expressions

A boolean expression combines properties of attributes and context:

```
boolexp  ->  <And> boolexp* </And> |
             <Or> boolexp* </Or> |
             <OneOf> boolexp* </OneOf> |
             <Not> boolexp* </Not> |
             <Imply> boolexp boolexp </Imply> |
             <Equiv> boolexp* </Equiv> |
             attributedescr |
             <Context> contextexp </Context>
```

Boolean expressions are used as

- constraints,
- guards in conditional constraints,
- guards in conditional content expressions (later...),
- guards in defaults (later...),
- points-to expressions (later...).

Attribute declarations and descriptions

An attribute **declaration** allows the presence of a given attribute:

```
attributedecl  ->  <AttributeDecl Name="attr.name" Optional="yes or no" ...>
                     stringtypeexp ...
                   </AttributeDecl>
```

("`...`" refers to *ID information* - described later...)

An attribute **description** tests the value of a declared attribute:

```
attributedescr  ->  <Attribute Name="attr.name" Value="attr.value"/> |
                    <Attribute Name="attr.name">
                      stringtypeexp?
                    </Attribute>
```

String types

A string type is a **regular expression** defining a set of Unicode text strings:

```
stringtypeexp  ->  <Sequence> stringtypeexp* </Sequence> |
                   <Optional> stringtypeexp </Optional> |
                   <ZeroOrMore> stringtypeexp </ZeroOrMore> |
                   <OneOrMore> stringtypeexp </OneOrMore> |
                   <Union> stringtypeexp* </Union> |
                   <Intersection> stringtypeexp* </Intersection> |
                   <Complement> stringtypeexp </Complement> |
                   <Repeat Value="..."> stringtypeexp </Repeat> |
                   <Empty/> |
                   <String Value="..."/> |
                   <CharSet Value="..."/> |
                   <CharRange Start=".." End=".."/> |
                   <AnyChar/>
```

String types are used to specify both valid **attribute values** and valid **character data**.

All reasonable data types can be specified using regular expressions. A few examples:

- time and date formats
- URI syntax
- social security numbers
- email addresses
- ISBN numbers

Content expressions

An element's content is viewed as a **sequence** of **element** nodes and **character data** nodes.

A content expression is a kind of regular expression on such sequences:

```
contentexp  ->  <Sequence> contentexp </Sequence> |
                <Optional> contentexp </Optional> |
                <ZeroOrMore> contentexp </ZeroOrMore> |
                <OneOrMore> contentexp </OneOrMore> |
                <Union> contentexp </Union> |
                <AnyElement/> |
                <Empty/> |
                <If> boolexp
                  <Then> contentexp </Then>
                  (<Else> contentexp </Else>)?
                </If> |
                stringtype |
                elementdescr
```

A *stringtype* here probes the contents of a character data node.

An *element descriptor* probes the **name** of an element node and assigns an **element ID** to it:

```
elementdescr  ->  <Element IDRef="element ID"/>
```

Evaluation is **eager** (e.g., `ZeroOrMore` consumes as much as possible).

Evaluating a stringtype expression or an element descriptor might insert **defaults** (-described later...).

Note: there is **no** intersection, complement, or setminus.

Multiple content descriptions

Unordered contents is a problem for regular expressions and may cause a combinatorial explosion.

Example:
"an `author` node must contain in sequence `first`, optionally `initial`, and then `last`, and somewhere in between, a `homepage` is required."

The DSD solution is **multiple descriptions** and **projected contents**.

Multiple content descriptions in a constraint

- are checked one at a time,
- only look at the contents projected onto the elements that they each mention,
- each consume the content nodes that they recognize, and
- at the end, each content node must be consumed once.

The following two content descriptions solve the problem in the example above:

```
<Content>
  <Sequence>
    <Element IDRef="first"/>
    <Optional><Element IDRef="initial"/></Optional>
    <Element IDRef="last"/>
  </Sequence>
</Content>

<Content>
  <Element IDRef="homepage"/>
</Content>
```

If the content sequence is	`first.homepage.initial.last.homepage`
then the first content description consumes	`first.........initial.last........`
and the second consumes	`.....homepage.....................`

so the sequence is rejected (because the second `homepage` is not consumed).

This allows concise specification of mixed ordered and unordered content.

Context patterns

- can be used to make defaults, constraints, and content descriptions **context dependent**.

(The *context* of a node is its sequence of ancestors.)

A context expression is a sequence of context terms:

```
contextexp  ->  contextterm*
```

It **matches** a context if the context can be decomposed into consecutive fragments, each matched by a context term.

A context term either matches an arbitrary fragment or a single element:

```
contextterm  -> <SomeElements/> |
                <Element (IDRef="element ID" | Name="element name")?>
                  attributedescr*
                </Element>
```

(Context expressions always implicitly start with a `<SomeElements/>`.)

DSD context patterns are reminiscent of CSS *selectors*.

Examples:

- XHTML `input` elements are only allowed inside a `form` element.
- In BookML, the `title` element is optional inside a `book` element with an `isbn` attribute.
- XHTML anchors are not allowed to be nested:

```
<Constraint>
  <Not>
    <Context>
      <Element Name="a"/><SomeElements/><Element Name="a"/>
    </Context>
  </Not>
</Constraint>
```

(This is placed as a constraint to `a`.)

Default insertion

- allows the processor to **insert missing attributes, elements, or chardata**
- often allows briefer XML documents

Declaration:

- defaults are **declared separately** from the structure definitions
- defaultability of elements must be **explicitly declared** (`Defaultable="yes"`)
- **cascading** defaults can be specified in the **application document**

Insertion:

- available defaults are **guarded by boolean expressions**
- insertion is **guided by the constraint checking** - defaults are inserted **on demand** by the constraints
- a notion of **specificity** determines the default when more than one is applicable

Default insertion is mixed into validation because of mutual dependence.

An example default definition

```
<Default>
  <Context>
    <Element Name="input">
      <Attribute Name="type" Value="text"/>
    </Element>
  </Context>
  <DefaultAttribute Name="length" Value="20"/>
</Default>
```

defines that the `length` of `input` fields of type `text` is by default `20`.

Application document defaults

- specifying default definitions in the **application document**.

```
<DSD:Default>
  <Context>
    <Element Name="form">
      <Attribute Name="action"/>
        <Sequence>
          <String Value="http://www.brics.dk/"/>
          <ZeroOrMore><AnyChar/></ZeroOrMore>
        </Sequence>
      </Attribute>
    </Element>
    <SomeElements/>
    <Element Name="input">
      <Attribute Name="type" Value="text"/>
    </Element>
  </Context>
  <DefaultAttribute Name="length" Value="30"/>
</DSD:Default>
```

(The length default previously defined is overridden for text type input elements inside form elements that have an action attribute whose value is a string starting with `http://www.brics.dk/`.)

An application document default

- is applicable for the subtree rooted by its parent element, and
- the inner-most application document default always overrides an outer one.

ID attributes and points-to requirements

- simple definitions and references inside a document.

A DSD can declare special application document attribute types:

- An **ID** attribute is a **definition** of its value (must be unique in that document).
- An **IDRef** attribute is a **reference** to the element containing the definition of the value.

(ID and IDRef originate from DTD.)

A DSD may also impose a **points-to** requirement on a reference: this is a boolean expression which must evaluate to true on the referenced node. This allows *semi-structured data* to be expressed.

Special attribute types are **declared** along with the attribute. Example:

```
<AttributeDecl ID="book-reference" IDType="IDRef">
  <PointsTo>
    <Context><Element Name="book"></Context>
  </PointsTo>
</AttributeDecl>
```

(a `book-reference` attribute has type IDRef and must refer to an attribute of type ID occuring in a `book` element.)

Redefinitions and evolving DSDs

- modularizing DSDs

Two issues:

- extending existing documents
- modifying existing definitions

Extending existing documents

DSD documents and application documents can be created as extensions of other documents using the `include` processing instruction:

```
<?include URI="..."?>
```

- the DSD processor **replaces** the processing instruction with the referenced document
- a document can only be included **once** into a given document (subsequent attempts are ignored)

Modifying existing definitions

Two extra attribute types:

RenewID

As ID but need not be unique: overrides previously occuring IDs and RenewIDs of same name.

CurrIDRef

IDRef refers to the *last* occuring ID or RenewID,
CurrIDRef refers to the *latest, previously occuring* not containing the CurrIDRef.

("previous", "last", etc., assumes obvious textual ordering in the XML document)

This mechanism is **used by the DSD language itself**.

Example:

Assume that in some existing DSD, a `book` element has been defined:

```
<ElementDef ID="book">
  <Constraint IDRef="book-constraint"/>
</ElementDef>

<ConstraintDef ID="book-constraint"> ... </ConstraintDef>
```

Now we want to **reuse** this DSD (using `include`), but **extend** the `book` constraint:

```
<ConstraintDef RenewID="book-constraint">
  <Constraint CurrIDRef="book-constraint"/>
  ...<i>the extension</i>...
</Constraint>
```

Note: no need to copy or change the contents of the *original* DSD!

Self-documentation

Documentation may be associated to most constructs in DSD:

`<Label> ... </Label>`

- can be used to attach a label to the construct

`<Doc> ... </Doc>`

- intended for full documentation

`<BriefDoc> ... </BriefDoc>`

- intended for a brief description

This allows a DSD to be virtually self-documenting towards application authors - and processors to provide useful help messages.

The Meta-DSD

DSD is self-describable: there is a DSD that **completely** captures the requirements for an XML document to be a valid DSD.

Such a Meta-DSD can be used

- as a **human readable description** of the DSD language to clarify details, and
- by DSD processors to **check whether a given XML document is a valid DSD**.

The official Meta-DSD can be found at `http://www.brics.dk/DSD/dsd.dsd`.

Application documents refer to their DSD using the `dsd` processing instruction in the document prolog:

```
<?dsd URI="URI of the DSD"?>
```

So DSDs usually contain the line

```
<?dsd URI="http://www.brics.dk/DSD/dsd.dsd"?>
```

An example: `article.dsd`

This example DSD for our notion of article documents (introduced in the XML section) illustrates a real-world (but incomplete) DSD.

It shows

- document inclusion (of "standard libraries")
- use of definitions and references and of inlined definitions
- mixed ordered and unordered contents (multiple content expressions)
- points-to requirements

and more...

This DSD has been developed "top-down". Several parts of the DSD are left for the energetic reader as an exercise :-)

(The file is also available at `http://www.brics.dk/DSD/examples/article.dsd`.)

Example: `article.dsd`, Part I

```
<?xml version="1.0" encoding="iso-8859-1"?>
<?dsd URI="http://www.brics.dk/DSD/dsd.dsd"?>

<!--this document is available at http://www.brics.dk/DSD/examples/article.dsd-->

<DSD IDRef="article" DSDVersion="1.0">

  <Title> A DSD description of the ArticleML syntax </Title>
  <Version> 0.3 </Version>
  <Author> Anders Møller </Author>

  <Doc>
    This example DSD is used in the XML tutorial; see
http://www.brics.dk/~amoeller/XML
    Note: the DSD is incomplete (and would benefit from some more Doc description
elements)
    Exercise to the reader: fill out the incomplete parts such that the DSD captures
all
    reasonable syntax requirements of ArticleML :-)
    This example covers much of the DSD language, however it does not
    illustrate redefinitions or default definitions.
  </Doc>

  <?include URI="http://www.brics.dk/DSD/library/standard_datatypes.dsd"?>
  <?include URI="http://www.brics.dk/DSD/library/standard_constraints.dsd"?>

  <ElementDef ID="article">
    <AttributeDecl Name="postscript"><StringType IDRef="URI"/></AttributeDecl>
    <AttributeDecl Name="pdf"><StringType IDRef="URI"/></AttributeDecl>
    <Sequence>
      <Element IDRef="article-title"/>
      <OneOrMore><Element IDRef="authors"/></OneOrMore>
      <Optional><Element IDRef="abstract"/></Optional>
      <OneOrMore><Element IDRef="section"/></OneOrMore>
      <Optional><Element IDRef="references"/></Optional>
      <Optional><Element IDRef="vitae"/></Optional>
    </Sequence>
  </ElementDef>
```

Example: `article.dsd`, Part II

```
  <ElementDef ID="article-title" Name="title">
    <StringType/>
  </ElementDef>

  <ElementDef ID="authors">
    <Sequence>
      <Element Name="names"><Content IDRef="simpletext"/></Element>
      <Optional><Element Name="affiliation"><Content
IDRef="simpletext"/></Element></Optional>
      <Optional><Element IDRef="link"/></Optional>
    </Sequence>
  </ElementDef>

  <ElementDef ID="abstract">
    <OneOrMore>
      <Element IDRef="paragraph"/>
    </OneOrMore>
  </ElementDef>

  <ElementDef ID="section">
    <AttributeDecl Name="id" IDType="ID" Optional="yes"/>
    <Element IDRef="section-title"/> <!--title element may be anywhere-->
    <Sequence>
      <ZeroOrMore><Element IDRef="paragraph"/></ZeroOrMore>
      <ZeroOrMore><Element IDRef="subsection"/></ZeroOrMore>
    </Sequence>
  </ElementDef>

  <ElementDef ID="section-title" Name="title"> <!--less restrictive than
article-title-->
    <Content IDRef="simpletext"/>
  </ElementDef>

  <ElementDef ID="subsection">
    <AttributeDecl Name="id" IDType="ID" Optional="yes"/>
    <Element IDRef="section-title"/>
    <ZeroOrMore><Element IDRef="paragraph"/></ZeroOrMore>
  </ElementDef>

  <ElementDef ID="references">
    <ZeroOrMore>
      <Element IDRef="reference-item"/>
    </ZeroOrMore>
  </ElementDef>
```

```
  <ElementDef ID="reference-item" Name="item">
    <AttributeDecl Name="id" IDType="ID"/>
    <Element Name="authors"><Content IDRef="simpletext"/></Element>
    <Element Name="title"><Content IDRef="simpletext"/></Element>
    <Optional><Element Name="publisher"><Content
IDRef="simpletext"/></Element></Optional>
    <Optional><Element Name="note"><Content
IDRef="paragraph-content"/></Element></Optional>
    <Optional><Element Name="month"><StringType/></Element></Optional>
    <Optional><Element Name="year"><StringType/></Element></Optional>
  </ElementDef>
```

Example: `article.dsd`, Part III

```
  <ElementDef ID="vitae">
    <Constraint IDRef="anything"/> <!-- INCOMPLETE -->
  </ElementDef>

  <ElementDef ID="link">
    <AttributeDecl Name="href" Optional="yes"><StringType
IDRef="URI"/></AttributeDecl>
    <Content IDRef="simpletext"/>
  </ElementDef>

  <ContentDef ID="simpletext">
    <ZeroOrMore>
      <Content IDRef="simpletext-part"/>
    </ZeroOrMore>
  </ContentDef>

  <ContentDef ID="simpletext-part">
      <Union>
        <StringType/>
        <Element IDRef="amp"/>
        <Element IDRef="nbsp"/>
        <!-- INCOMPLETE -->
      </Union>
  </ContentDef>

  <ContentDef ID="paragraph-content">
    <ZeroOrMore>
      <Union>
        <Content IDRef="simpletext-part"/>
        <Element IDRef="cite"/>
        <Element IDRef="ref"/>
        <Element IDRef="link"/>
        <Element IDRef="tt"/>
        <Element IDRef="it"/>
        <Element IDRef="bf"/>
        <Element IDRef="br"/>
        <Element IDRef="itemize"/>
```

```
      <Element IDRef="oneline"/>
      <Element IDRef="example"/>
      <!-- INCOMPLETE -->
    </Union>
  </ZeroOrMore>
</ContentDef>

<ElementDef ID="paragraph" Name="p">
  <Content IDRef="paragraph-content"/>
</ElementDef>
```

Example: `article.dsd`, Part IV

```
  <ElementDef ID="amp"/>

  <ElementDef ID="nbsp"/>

  <ElementDef ID="percent"/>

  <ElementDef ID="br"/>

  <ElementDef ID="tt">
    <Content IDRef="paragraph-content"/>
  </ElementDef>

  <ElementDef ID="it">
    <Content IDRef="paragraph-content"/>
  </ElementDef>

  <ElementDef ID="bf">
    <Content IDRef="paragraph-content"/>
  </ElementDef>

  <ElementDef ID="itemize">
    <Constraint IDRef="anything"/> <!-- INCOMPLETE -->
  </ElementDef>

  <ElementDef ID="oneline">
    <Constraint IDRef="anything"/> <!-- INCOMPLETE -->
  </ElementDef>

  <ElementDef ID="example">
    <Constraint IDRef="anything"/> <!-- INCOMPLETE -->
  </ElementDef>

  <ElementDef ID="cite">
    <AttributeDecl Name="article" IDType="IDRef">
      <PointsTo><Context><Element IDRef="reference-item"/></Context></PointsTo>
    </AttributeDecl>
  </ElementDef>

  <ElementDef ID="ref">
    <AttributeDecl Name="section" IDType="IDRef" Optional="yes">
      <!-- INCOMPLETE, use PointsTo -->
    </AttributeDecl>
    <AttributeDecl Name="subsection" IDType="IDRef" Optional="yes">
      <!-- INCOMPLETE, use PointsTo -->
    </AttributeDecl>
    <OneOf><Attribute Name="section"/><Attribute Name="subsection"/></OneOf>
  </ElementDef>

</DSD>
```

Implementation

DSD processor

- written in C, uses the Expat parser, runs on UNIX and Windows
- is available in Open Source distribution
- experimental prototype
- guarantees linear-time processing of XML documents (the constant depends on the DSD)
- bootstrapped using Meta-DSD (simplifies implementation a lot!)
- self-application of 450 line meta-DSD takes 1/2 sec.
- gives usable error-messages when the given application document does not conform to its DSD

See `www.brics.dk/DSD/implementation.html` for more information...

Other proposals

XML Schema

- developed by W3C Working Group (52 people!)
- currently a Working Draft
- divided into "Part 1: Structures" and "Part 2: Datatypes"

XML-Data, **DCD**, **SOX**, and **DDML**

- earlier proposals (W3C Notes)
- predecessors of XML Schema

Schematron

- based on XPath tree patterns (instead of grammars)

Unique features of DSD:

- conditional constraints
- element IDs as non-terminals
- constraints on reference targets (points-to requirements)
- context-based default insertion
- simple redefinitions
- linear-time implementation

Status

Future DSD issues:

- namespace support
- better white-space handling
- global constraints (extra constraints checked separately after normal processing)
- object-oriented inheritance mechanism

- will be included in DSD 1.1.

Is there room for more than one XML schema language?
- if not, let's hope for the simplest and most expressive to win :-)

Links to more information

www.brics.dk/DSD

The DSD homepage, contains the specification, examples, implementation, and more...

www.w3.org/TR/xmlschema-0

W3C's XML Schema Working Draft, Part 0: Primer

www.w3.org/TR/xmlschema-1

W3C's XML Schema Working Draft, Part 1: Structures

www.w3.org/TR/xmlschema-2

W3C's XML Schema Working Draft, Part 2: Datatypes

www.w3.org/TR/NOTE-xml-schema-req

XML Schema Requirements (W3C Note)

www.oasis-open.org/cover/schemas.html

Robin Cover's XML schema information

www.ascc.net/xml/resource/schematron/schematron.html

The Schematron language

www.xml.com/pub/1999/12/dtd

An xml.com article on Schema languages

XLink, XPointer, and XPath

Linking and addressing

(Prerequisites: XML, Namespaces)

XLink, XPointer, and XPath

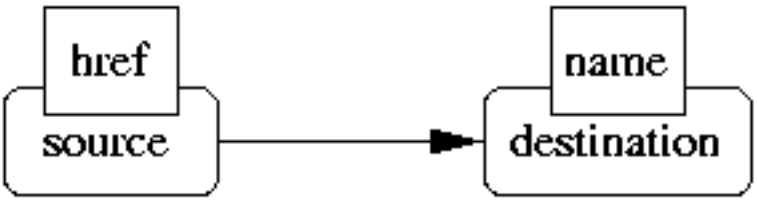
- imagine a Web without links...

Three layers:

- **XLink**
  - a generalization of the HTML link concept
  - higher abstraction level (intended for general XML - not just hypertext)
  - more expressive power (multiple destinations, special behaviours, out-of-line links, ...)
  - uses XPointer to locate resources
- **XPointer**
  - an extension of XPath suited for linking
  - specifies connection between XPath expressions and URIs
- **XPath**
  - a declarative language for locating nodes and fragments in XML trees
  - used in both XPointer (for addressing) and XSL (for pattern matching)

Problems with HTML links

The HTML link model:



Problems when using the HTML model for general XML:

Link syntax and semantics is not defined separately from rest of HTML:

- In HTML, links are recognized by element names (`A`, `IMG`, ..)
  - we want a *generic XML solution*.
- The "semantics" of a link is completely defined in the higher-level specification
  - we want control of general semantic features, e.g. of link *actuation*.

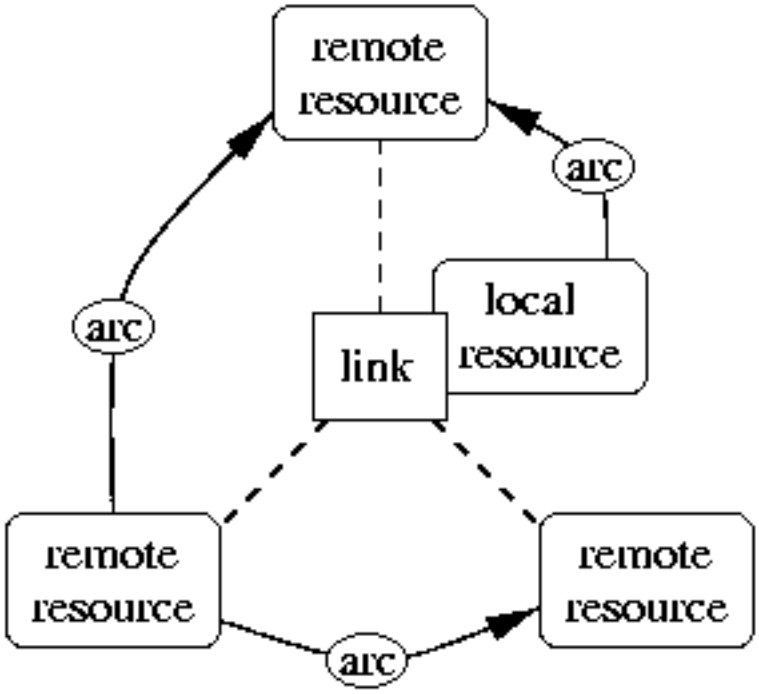
HTML links are just too simple:

- An anchor must be placed at every link destination (problem with read-only documents)
  - we want to express *relative locations* (XPointer!).
- The link definition must be at the same location as the link source
  - we want *out-of-line* links ("link databases").
- Only individual nodes can be linked to
  - we want links to whole *tree fragments*.
- A link always has one source and one destination
  - we want links with *multiple sources and destinations*.
- Links are anonymous
  - we want human-readable *labels*.

The XLink linking model

Basic XLink terminology:

Link: Explicit relationship between two or more resources.
Linking element: An XML element that asserts the existence and describes the characteristics of a link.
Locator: An identification of a remote resource that is participating in the link.



- one **linking element** defines a set of traversable **arcs** between some **resources**.

A **local resource** comes from the linking element's own content.

Out-of-line link: link with no local resource.
Inline link: link with local resource.

An example

- a linking element defining an out-of-line extended link involving two remote resources

```
<extended>
  <locator href="#Fred" role="student"/>
  <locator href="teachers.xml#Joe" role="teacher"/>
  <arc from="student" to="teacher" show="embed"/>
</extended>
```

- `locator` locates a remote resource
- `arc` defines traversal rules
- `show="embed"` defines link behaviour

Recognizing XLink

- the XLink namespace

XLink can be used in any markup language. We need a general method for recognizing XLink information.

XLink information is recognized using the namespace

`http://www.w3.org/1999/xlink/namespace/`

- the namespace often (but not necessarily) uses namespace prefix `xlink`
- elements and attributes *not* belonging to this namespace are ignored by XLink processors

Linking elements

- defining links

- a general **linking element** is defined using an `extended` element (which can contain the following)
- a **local resource** is defined using a `resource` element
- a **remote resource** is defined using a `locator` element with a `href` attribute (an XPointer expression locating the resource)
- **arcs** (traversal rules) are defined using `arc` elements
  - an `arc` element has a `from` and a `to` attribute
  - both `resource` and `locator` have have a `role` attribute
  - the `arc` element defines a set of arcs: from each resource having the `from` role to each resource having the `to` role
  - if `from` or `to` is omitted, any role matches

(Note the unfortunate terminology: a "resource" is defined either by a `resource` element or by a `locator` element)

Both `extended` and `locator` can have `title` attributes or `title` sub-elements, providing **human-readable titles** for links and resources.

XPointer is described later - just think of XPointer expression as URIs for now...

Behaviour

- link semantics

Arcs can be annotated with abstract behaviour information using the following `arc` attributes:

`show` - **what happens when the link is activated?**

Possible values:

`embed`

insert the target resource (the one at the end of the arc) immediately after the display of the source resource (the one at the beginning of the arc, where traversal was initiated) (example: as images in HTML)

`new`

display the target resource some other place without affecting presentation of the resource from which traversal was initiated (example: as `target="_new"` in an HTML link)

`replace`

replace the presentation of the resource containing the linking element with a presentation of the one being linked to (example: as normal HTML links)

`undefined`

behaviour specified elsewhere

`actuate` - **when is the link activated?**

Possible values:

`onLoad`

traverse the link immediately when recognized (example: as HTML images)

`onRequest`

traverse when explicitly requested (example: as normal HTML links)

`undefined`

behaviour specified elsewhere

Note: these notions of link behaviour are rather abstract and do not make sense for all applications.

Links in non-XLink elements

Linking elements can be mixed into non-XLink elements.

As an alternative to writing

```
<xlink:extended ...> ... </xlink:extended>
```

one can write

```
<foo xlink:type="extended" ...> ... </foo>
```

(where `foo` is some non-XLink element)

- similarly for other XLink elements: `locator`, `arc`, `resource`, `title`, `simple`

Simple vs. Extended links

- for compatibility and simplicity

Two kinds of links:

- **extended** - the general ones we have seen so far
- **simple** - a restricted version of extended links: only for **two-ended inline** links (enough for HTML-style links)

Convenient shorthand notation for simple links:

```
<xlink:simple href="..." show="..." actuate="..."/>
```

which is equivalent to:

```
<xlink:extended>
  <xlink:resource role="local"/>
  <xlink:locator href="..." role="remote"/>
  <xlink:arc from="local" to="remote" show="..." actuate="..."/>
</xlink:extended>
```

(`show` and `actuate` are optional.)

Using schema default mechanisms with XLink

- many XLink properties can conveniently be specified as defaults

Example:

One may write

```
<A xlink:href="..."/>
```

instead of

```
<A xlink:type="simple" xlink:href="..."
   xlink:show="replace" xlink:actuate="onRequest"/>
```

if proper defaults have been defined for the A element by the schema.

In DSD, this could be done by:

```
<Default>
  <Context>
    <Element Name="A"><Attribute Name="href"/></Element>
  </Context>
  <DefaultAttribute Name="xlink:type" Value="simple"/>
  <DefaultAttribute Name="xlink:show" Value="replace"/>
  <DefaultAttribute Name="xlink:actuate" Value="onRequest"/>
</Default>
```

(The DSD language is the topic of a later section.)

External link sets

linkbase: an XML document whose primary purpose is to contain XLink link elements defining out-of-line links.

Linkbases are located either

- in some out-of-band way (using some external source), or
- in-band using *external linksets*.

An **external linkset** is a link with role `xlink:external-linkset`, e.g.

```
<xlink:extended role="xlink:external-linkset">
  <xlink:locator href="..."/>
</xlink:extended>
```

When processing a document with an external linkset, all links in the linkbase being referenced to should be fetched and processed.

(Both extended and simple links can have `role` attributes - *link* roles have nothing to do with *resource* roles.)

XPointer: Why, what, and how?

- relative addressing: allows links to places with no anchors
- can point to substrings in character data and to whole tree fragments (node ranges)
- used by XLink to locate remote link resources

Example of an XPointer:

```
                                   URI
 -------------------------------------------------------------------
/                                                                   \
http://www.foo.org/bar.xml#xpointer(article/section[position() &lt;= 5])
                           |         \                              / |
                           |           -----------------------------   |
                            \                   XPointer expression    /
                             \                                        /
                              ----------------------------------------
                                          fragment identifier
```

(points to the first five section elements in the article root element.)

In HTML, fragment identifiers may denote anchor IDs -- XPointer generalizes that.

XPointer vs. XPath

- XPointer is based upon XPath

- an XPath is evaluated wrt. some **context**; XPointer specifies this context
- XPath says nothing about **URIs**; XPointer specifies that connection
- XPointer **adds some features** not available in XPath

XPointer fragment identifiers

An *XPointer fragment identifier* (the substring to the right of # in the URI) is either

- the value of some **ID** attribute in the document (ID attributes are specified by the DTD or DSD),
- a sequence of element numbers denoting the **path** from the root to an element (e.g. `/1/27/3`), or
- a sequence of the form

  ```
  xpointer(...) xpointer(...) ...
  ```

  containing a list (typically of length 1) of **XPointer expressions**.

  Each expression is evaluated in turn, and the first where evaluation succeeds is used. (This allows alternative pointers to be specified thereby increasing robustness.)

(XPointer allows for other means of specifying pointers - XPointer pointers are recognized by "`xpointer(...)`".)

Next: We will now dig into **XPath** and then later describe what additional features XPointer adds to XPath...

XPath: Location paths

XPath is a declarative language for addressing (used in XPointer) and pattern matching (used in XSLT).

The central construct is the **location path**, which is a sequence of **location steps** separated by `/`.

- A **location step** is evaluated wrt. some context resulting in a set of nodes.
- A **location path** is evaluated *compositionally*, *left-to-right*, starting with some initial context.

  Each node resulting from evaluation of one step is used as context for evaluation of the next, and the results are unioned together.

A **context** consists of:

- a context **node**,
- a context **position** and **size** (two integers, 1<=position<=size),
- **variable bindings**, a **function library**, and a set of **namespace declarations**.

Initial context: defined externally (e.g. by XPointer or XSLT).

(Location paths starting with `/` always use the document root as initial context node!)

Example:

```
child::section[position()<6]/descendant::cite/attribute::href
```

selects all `href` attributes in `cite` elements in the first 5 `section`s of an article.

Location steps

A **location step** has the form

axis `::` *node-test* `[` *predicate* `]`

- The **axis** selects a set of *candidate nodes* (e.g. the child nodes of the context node).
- The **node-test** performs an **initial filtration** of the candidates based on their
  - *types* (chardata node, processing instruction, etc.), or
  - *names* (e.g. element name).
- The **predicates** (zero or more) cause a further, potentially more complex, filtration. Only candidates for which the predicates evaluate to *true* are kept.

The candidates that survive the filtration constitute the *result*.

This structure of location paths and steps makes implementation rather easy.

The example from before:

```
child::section[position()<6]/descendant::cite/attribute::href
```

selects all `href` attributes in `cite` elements in the first 5 `section`s of an article.

Axes

Available axes:

`child`	the children of the context node
`descendant`	all descendants (children, childrens children, ...)
`parent`	the parent (empty is at the root)
`ancestor`	all ancestors from the root to the parent
`following-sibling`	siblings to the right
`preceding-sibling`	siblings to the left
`following`	all following nodes in the document
`preceding`	all preceding nodes in the document
`attribute`	the attributes of the context node
`namespace`	namespace declarations in the context node
`self`	the context node itself
`descendant-or-self`	the union of `descendant` and `self`
`ancestor-or-self`	the union of `ancestor` and `self`

Note that **attributes** and **namespace declarations** are considered a special kind of nodes.

Some of these axes assume a **document ordering** of the tree nodes. The ordering essentially corresponds to a **left-to-right preorder** traversal of the document tree.

The resulting sets are **ordered** intuitively, either **forward** (in document order) or **reverse** (reverse document order).
For instance, `following` is a forward axis, and `ancestor` is a reverse axis.

Node tests

Testing by node **type**:

`text()`	chardata nodes
`comment()`	comment nodes
`processing-instruction()`	processing instruction nodes
`node()`	all nodes (not including attributes and namespace declarations)

Testing by node **name**:

`name`	nodes with that name
`*`	any node

When testing by name, only nodes of the "axis principal node type" are considered (attributes for `attribute` axis, namespace nodes for `namespace` axis, element nodes for all other axes).

Predicates

- expressions coerced to type *boolean*

A predicate filters a node-set by evaluating the predicate expression on each node in the set with

- that node as the **context node**,
- the size of the node-set as the **context size**, and
- the position of the node in the node-set wrt. the axis ordering as the **context position**.

Expressions

Available types: **node-set** (set of nodes), **boolean** (true or false), **number** (floating point), **string** (Unicode text).

Abstract syntax for expressions:

```
exp -> $variable
     | ( exp )
     | literal
     | numeral
     | function ( arguments )
     | boolean-expression
     | numerical-expression
     | node-set-expression
```

Coercion may occur at function arguments and when expressions are used as predicates.

Variables and functions are evaluated using the context.
Boolean, numerical, and node-set expressions are described in the following...

Operators

Boolean expressions:

Available boolean operators:

`or`, `and`, `=`, `!=`, `<`, `>`, `<=`, `>=`

Standard precedence, all left associative.

Numerical expressions:

Available numerical operators:

`+`, `-`, `*`, `div`, `mod`

Node-set expressions:

- location paths, filtered by predicates.

Node-set expression operators:

`|` (node-set union)

Core function library

Node-set functions:

last()	returns the context size
position()	returns the context position
count(*node-set*)	returns number of nodes in node-set
name(*node-set*)	return string representation of first node in node-set
...	...

String functions:

string(*value*)	type cast to string
concat(*string*, *string*, ...)	string concatenation
...	...

Boolean functions:

boolean(*value*)	type cast to boolean
not(*boolean*)	boolean negation
...	...

Number functions:

number(*value*)	type cast to number
sum(*node-set*)	sum of number value of each node in node-set
...	...

- see the XPath specification for the complete list.

Abbreviations

Syntactic sugar: convenient notation for common situations

Normal syntax	***Abbreviation***
`child::`	*nothing* (so `child` is the default axis)
`attribute::`	`@`
`/descendant-or-self::node()/`	`//`
`self::node()`	`.` (useful because location paths starting with `/` begin evaluation at the root)
`parent::node()`	`..`

Example:

```
.//@href
```

selects all `href` attributes in descendants of the context node.

Furthermore, the **coercion rules** often allow compact notation, e.g.

```
foo[3]
```

refers to the third `foo` child element of the context node (because `3` is coerced to `position()=3`).

XPointer: Context initialization

An XPointer is basically an XPath expression occuring in a URI.

When evaluated, the **initial context** is defined as follows:

- The **context node** is the root node of the document.
- The **context position** and **size** are both 1.
- The **variable bindings** are empty.
- The **function library** consists of the core XPath functions + a few extra functions
- The **namespace declarations** are chosen to be the ones whose scope contains the XPointer.

Extra XPointer features

XPointer provides a more **fine-grained addressing** than XPath.

- Instead of just *nodes*, XPointers address **locations**, which can be *nodes*, *points*, or *ranges*.
- A **point** can represent the location preceding or following any *individual character* in a chardata node.
  The special node test
  `point()`
  selects the set of points of a node.
- A **range** consists of two points in the same document,
  and is specified using a *range expression* of the form
  `expr to expr`.
- XPointer provides some **extra functions**:

`here()`	get location of element containing current XPointer
`origin()`	get location where user initiated link traversal
`start-point(location-set)`	get start point of location set
`string-range(...)`	find matching substrings
`...`	

Example 1:

```
/descendant::text()/point()[position()=0]
```

selects the first character of all character data nodes in the document.

Example 2:

```
/section[1] to /section[3]
```

selects everything from the beginning of the first `section` to the end of the third.

Tools

Kinds of tools supporting XLink:

- browsers
- parsers
- link bases

www.fujitsu.co.jp/hypertext/free/HyBrick/en

the HyBrick browser

www.loria.fr/projets/XSilfide/EN/sxp

the SXP parser

www.stepuk.com/x2x/x2x_ove.asp

the X2X link base

pages.wooster.edu/ludwigj/xml

the Link browser

Warning: most tools do *not* support the newest specifications.

The `article` example continued

Some applications of XLink, XPointer, and XPath for `article` documents:

1. **document inclusion**

   Using `show="embed"`, one could divide `article.xml` into a document for each section plus one combining the others using XLinks.

2. **section references**

   A very typical use of XLinks is to create HTML-like links inside a document, e.g. from section references to sections (as `label` and `ref` in LaTeX).

3. **post-it notes**

   Using *out-of-line links* and a smart browser, a group of people can annotate an article with "post-it notes" for discussion - without having write-access to the document. They simply need to agree on a set of URIs to XLink linkbases defining the annotations. The smart XLink-aware browser lets them select portions of the article (as XPointer ranges), comment the portion by creating an XLink to a small XHTML document, view each others comments, even place comments on comments, and perhaps also aid in structuring the comments.

4. **index in external linkbase**

   Using *external linkbases*, the index section of an article can conveniently be specified separately from the article itself. The article just contains an `xlink:external-linkset` link to the document defining the index.

5. **cross-references between articles and article bases**

   The reference section of an article could contain XLinks to other article documents. These links may be collected in large article bases.

Links to more information

www.w3.org/TR/xlink

W3C's Working Draft on XLink

www.w3.org/TR/xptr

W3C's Candidate Recommendation on XPointer

www.w3.org/TR/xpath

W3C's Candidate Recommendation on XPath

www.stg.brown.edu/~sjd/xlinkintro.html

a brief introduction to XML linking

metalab.unc.edu/xml/books/bible/updates/16.html

a chapter from The XML Bible on XLink

metalab.unc.edu/xml/books/bible/updates/17.html

a chapter from The XML Bible on XPointer (and XPath)

XSLT - XSL Transformations

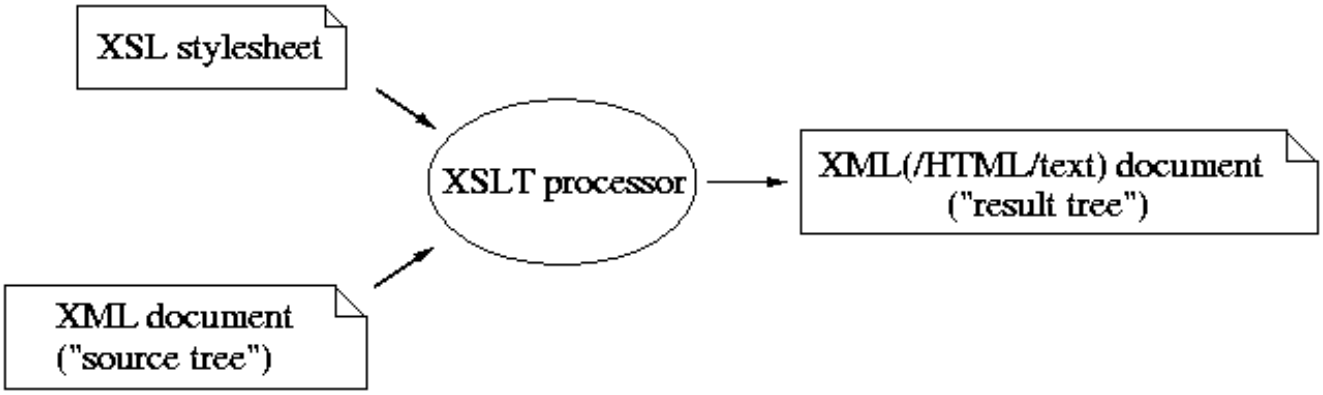
(Prerequisites: XML, Namespaces, XPath)

XSLT - XSL Transformations

XSL (eXtensible Stylesheet Language) consists of two parts: *XSL Transformations* and *XSL Formatting Objects*.

- An **XSLT stylesheet** is an XML document defining a **transformation** for a class of XML documents.
- A stylesheet seperates **contents and logical structure** from **presentation**.
- **Not** intended as completely general-purpose XML transformation language - designed for **XSL Formatting Objects**. Nevertheless: XSLT is generally useful.

The basic idea:



The basic design:
XSLT is **declarative** and based on **pattern-matching and templates**.

Usage scenarios

XSL transformations can be used

- on **server side** (pre-processing), e.g. using LotusXSL
- on **client side** (on-the-fly processing), e.g. using IE5

to transform XML documents from one markup-language to another.

Examples:

- ArticleML --> XHTML (viewable by browsers understanding XHTML)
- ArticleML --> HTML (the backwards compatible way of using XSLT)
- ArticleML --> LaTeX (a non-standard, but possible, use of XSLT)
- ArticleML --> XSL Formatting Objects (a typical future use of XSLT)
- ArticleML --> ArticleML (the source and target language can be the same)
- DSD --> HTML (see `http://www.brics.dk/DSD/dsd2html.html`)
- XHTML --> XSL Formatting Objects (likely, a typical future standard XSL stylesheet)
- ...

(ArticleML is the class of XML documents defined by `article.dsd` in the section on DSD.)

XSL Formatting Objects is the "second half" of XSL. It is an *XML vocabulary* for specifying formatting in a more low-level and detailed way than possible with HTML+CSS.

XSL can be introduced gently to the Web world, since it both can translate into HTML (which every browser understands), and XSL F.O. (which future browers will understand).

Processing model

template rule = pattern + template

Construction of result tree fragment:

- the **source tree** is processed by processing the root
- a single **node** is processed by
    1. **finding** the template rule with the best matching pattern
    2. **instantiating** its template (creates fragment + continues processing recursively)
- a **node list** is processed by processing each node in order

current node: the node currently being processed
current node list: the node list currently being processed
(used for *evaluation context* later)

Structure of a style sheet

An XSL stylesheet is itself an XML document:

```
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0"
                xmlns="...">
    .
    .
    .
  <xsl:template match="pattern">   \
    template                        > a template rule
  </xsl:template>                  /
    .
    .                     <- other top-level elements
    .
</xsl:stylesheet>
```

The namespace `http://www.w3.org/1999/XSL/Transform` is used to recognize the XSL elements; elements from other namespaces constitute *literal result fragments*.

A tiny example

Recall the **business card** example from the DSD section:

```
<card type="simple">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 456-1414</phone>
</card>
```

We define an **XHTML rendering semantics** for our business-card markup language using an XSLT stylesheet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
                xmlns="http://www.w3.org/1999/xhtml">

  <xsl:template match="card[@type='simple']">
    <html xmlns="http://www.w3.org/1999/xhtml">
      <title>business card</title><body>
        <xsl:apply-templates select="name"/>
        <xsl:apply-templates select="title"/>
        <xsl:apply-templates select="email"/>
        <xsl:apply-templates select="phone"/>
    </body></html>
  </xsl:template>

  <xsl:template match="card/name">
    <h1><xsl:value-of select="text()"/></h1>
  </xsl:template>

  <xsl:template match="email">
    <p>email: <a href="mailto:{text()}"><tt>
      <xsl:value-of select="text()"/>
    </tt></a></p>
  </xsl:template>

  ...
</xsl:stylesheet>
```

Resulting document:

```
<html xmlns="http://www.w3.org/1999/xhtml"><title>business card</title>
<body><h1>John Doe</h1><h3><i>CEO, Widget Inc.</i></h3>
<p>email: <a href="mailto:john.doe@widget.com"><tt>john.doe@widget.com</tt></a></p>
<p>phone: (202) 456-1414</p>
</body></html>
```

A browser might show this as:

John Doe

CEO, Widget Inc.

email: john.doe@widget.com

phone: (202) 456-1414

Patterns

- simple **XPath** expressions evaluating to **node-sets**.

A node **matches** a pattern if the node is member of the result of **evaluating** the pattern wrt. **some** context.
(Compare this with XPointer use of XPath.)

Typical structure:

pattern:	*location path* **`|`** ... **`|`** *location path*
location path:	`/`*step*`/` ... `//` ... `/`*step*
step:	*axis nodetest predicate*

Example:

```
match="section/subsection|appendix//subsection"
```

matches `subsection` elements occuring either as child elements of `section` elements or as descendants of `appendix` elements.

More precisely:

- a pattern is a set of XPath location paths separated by `|` (union)
- the axes are restricted to `child` (default) and `attribute` (@)
- the location paths may start with `id(..)` or `key(..)`

(The whole XPath language is used for *expressions* later...)

Templates

- literal result fragments
- recursive processing
- computed result fragments
- conditional processing
- sorting
- variables and parameters
- numbering
- keys

Literal result fragments

text + non-XSL namespace elements

considered literal fragments (constants)

`<xsl:text ...> ... </...>`

as raw text but allows control of white-space stripping and escaping

`<xsl:comment> ... </...>`

creates comment

Notice that since literal fragments are part of the stylesheet XML document, only well-formed XML will be generated (unless disabling output escaping).

Recursive processing

`<xsl:apply-templates select="`*`node-set expression`*`" .../>`

apply pattern matching and template instantiation on selected nodes (default: all children);
`mode="..."` on `xsl:template` and `xsl:apply-templates` allows an element to be processed multiple times in different ways

`<xsl:call-template name="..."/>`

invoke template by name (`name="..."` on `xsl:template`)

`<xsl:for-each select="`*`node-set expression`*`"> `*`template`*` </...>`

instantiate template for each node in node-set (document order by default)

The value of a `select` attribute is basically an **XPath expression** evaluated with the *current node* and *current node list* as context.

Computed result fragments

`<xsl:element name="..." namespace="..."> ... </...>`

computed element (attributes are *attribute value templates*)

`<xsl:attribute name="..." namespace="..."> ... </...>`

computed attribute (inside `xsl:element`)

`<xsl:value-of select="..."/>`

computed text (expression converted to string)

`<xsl:processing-instruction name="..."> ... </...>`

computed processing instruction

`<xsl:copy> ... </...>`

copy current node (+namespace nodes) and apply template

`<xsl:copy-of select="..."> ... </...>`

copy selected nodes and apply template

`<xsl:attribute-set name="..."> ... </...>`

define named set of attributes, used by `xsl:use-attribute-sets` in literal result elements, `xsl:copy`, or `xsl:attribute-set`

Attribute value templates: may contain `{`*`expression`*`}`, an **XPath expression** which is evaluated (and coerced to string) on instantiation.

Conditional processing

```
<xsl:if test="expression"> ... </...>
```

apply template if expression (converted to boolean) evaluates to true

```
<xsl:choose>
  <xsl:when test="expression"> ... </...>
  ...
  <xsl:otherwise> ... </...>
</...>
```

test conditions in turn, apply template for first true

Sorting

- choosing order for `xsl:apply-templates` and `xsl:for-each` (default: document order)

`<xsl:sort select="`*`expression`*`" .../>`

> sequence of `xsl:sort` placed in `xsl:apply-templates` or `xsl:for-each` defining lexicographic order (first occuring: primary key, etc.) - expression evaluated on each node in node-set, result converted to string for comparison

Extra attributes:

`order="ascending/descending"`

`lang="..."`

`data-type="text/number"`

`case-order="upper-first/lower-first"`

Variables and parameters

- for reusing results of computations and parameterizing templates and whole stylesheets

- static scope rules
- can hold any XPath value (string, number, boolean, node-set) + **result-tree fragment**
- purely declarative: variables cannot be updated
- can be global or local to a template rule

Declaration:

`<xsl:variable name="..." select="`*`expression`*`"/>`

variable declaration for normal XPath value

`<xsl:variable name="...">` *`template`* `</..>`

variable declaration for result tree fragment (template is instantiated to give value)

`<xsl:param name="..." select="`*`expression`*`"/>`

parameter declaration for normal XPath value (the value is the *default value* of the parameter)

`<xsl:param name="...">` *`template`* `</..>`

parameter declaration for result tree fragment

Use:

`$`*`name`*

returns XPath value in expressions, e.g. attribute value templates (recall that XPath context contains *variable bindings*)

`<xsl:with-param name=".." select="..."/>` and
`<xsl:with-param name="...">` *`template`* `</..>`

passes parameters in `xsl:call-template` or `xsl:apply-templates`

Note: result tree fragments held by variables or parameters can be used as source in pattern matching and template instantiation!

Numbering

```
<xsl:number value="expression"        converted to number
            format="..."              default: 1.

            level="..."               any/single/multiple
            count="..."               select what to count
            from="..."                select where to start counting

            lang="..."
            letter-value="..."
            grouping-separator="..."
            grouping-size="..."/>
```

- If `value` is specified, that value is used.
- Otherwise, the action is determined by `level`:
  - `level="any"`: number of preceding `count` nodes occuring after `from`
    (example use: numbering footnotes)
  - `level="single"` (the default): as `any` but only considers ancestors and their siblings
    (example use: numbering ordered list items)
  - `level="multiple"`: generates whole list of numbers
    (example use: numbering sections and subsections at the same time)

- lots and lots of details omitted here...

Keys

- advanced node IDs

a key is a triple (**node**, **name**, **value**)

`<xsl:key match="`*`pattern`*`" name="..." use="`*`node set expression`*`"/>`

declares set of keys - one for each node matching the pattern and for each node in the node set

Comparison to DTD (or DSD) IDs:

- keys are declared in the stylesheet (not in the DTD)
- keys allow different "name spaces"
- key values can be placed anywhere (not just as attributes)
- one node may have several keys
- keys need not be unique

Extra XPath key functions:

`key(`*`name expression, value expression`*`)`

returns nodes with given key name and value

`key(`*`name expression, node-set expression`*`)`

returns union of `key(`*`name, string value of node`*`)` for each node in node-set

These are often used together with:

`generate-id(`*`singleton node-set expression`*`)`

returns unique string identifying the given node

An example

We will now use XSLT to define a transformation from our article-class of XML documents (see `article.dsd`) into HTML.

This XSLT example illustrates

- an almost-complete real-world application of XSLT
- most features of XSLT programming

In particular, it shows how to use XSLT to automatically generate section numbering and references.

It is left for the reader as an exercise to add automatic sorting, numbering, and referencing for article citation references.

The complete stylesheet:	`article2html.xsl`
The original XML article document:	`article.xml`
The resulting HTML article document:	`article.html`

Some selected snippets coming up...

Example: `article2html.xsl`, Part I

The first part of the stylesheet:

```
<?xml version="1.0"?>
<!--
  This is a small example XSLT stylesheet transforming ArticleML documents into HTML.
  The example is incomplete: important things like citation references
  are still missing. (Exercise to the reader: use xsl:variable,
  xsl:sort, and xsl:number to finish the parts about citation references.)
-->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:variable name="style">my-style</xsl:variable>

<xsl:output method="html" indent="no"/>

<xsl:key name="idkey" match="*[@id]" use="@id"/>

<!-- MAIN TEMPLATE -->

<xsl:template match="article">
<html>
  <head>
    <title><xsl:apply-templates select="title" mode="raw"/></title>
    <xsl:call-template name="{$style}"/>
  </head>
  <body>
    <xsl:call-template name="header"/>
    <h1><xsl:apply-templates select="title" mode="raw"/></h1>
    <xsl:apply-templates select="authors"/>
    <xsl:apply-templates select="abstract"/>
    <xsl:apply-templates select="section"/>
    <xsl:apply-templates select="references"/>
    <xsl:apply-templates select="vitae"/>
  </body>
</html>
<xsl:text>
</xsl:text>
<xsl:comment>this HTML page was generated by article2html.xsl</xsl:comment>
</xsl:template>
```

Example: `article2html.xsl`, Part II

Transforming the abstract, sections, subsections, and references:

```
<xsl:template match="abstract">
<table width="80%" align="center"><tr><td>
<h4>Abstract</h4>
<xsl:apply-templates/>
</td></tr></table>
</xsl:template>

<xsl:template match="section">
<h3><a name="{generate-id()}">
  <xsl:number format="1. "/>
  <xsl:apply-templates select="title" mode="raw"/>
</a></h3>
<xsl:apply-templates/>
</xsl:template>

<xsl:template match="subsection">
<h4><a name="{generate-id()}">
  <xsl:number level="multiple" count="section|subsection" format="1.1 "/>
  <xsl:apply-templates select="title" mode="raw"/>
</a></h4>
<xsl:apply-templates/>
</xsl:template>

<xsl:template match="ref[@section]">
<a href="#{generate-id(key('idkey',@section))}">
  <xsl:for-each select="key('idkey',@section)">
    <xsl:number level="single" count="section" format="1"/>
  </xsl:for-each>
</a>
</xsl:template>

<xsl:template match="ref[@subsection]">
<a href="#{generate-id(key('idkey',@subsection))}">
  <xsl:for-each select="key('idkey',@subsection)">
    <xsl:number level="multiple" count="section|subsection" format="1.1"/>
  </xsl:for-each>
</a>
</xsl:template>
```

Example: `article2html.xsl`, Part III

Transforming example code and special characters:

```
<xsl:template match="example">
<p/><br/>
<table border="1" cellpadding="5" width="100%" bgcolor="#f5dcb3"><tr><td><pre>
<xsl:apply-templates/>
</pre></td></tr></table>
<br/>
</xsl:template>

<xsl:template match="percent">
<xsl:text>%</xsl:text>
</xsl:template>

<xsl:template match="nbsp">
<xsl:text disable-output-escaping="yes">&amp;nbsp;</xsl:text>
</xsl:template>
```

Issues not covered

- conflict resolution (`priority`)
- output modes (`xml`, `html`, `text`)
- white-space handling (`strip-space`, `preserve-space`)
- output escaping (`disable-output-escaping`)
- additional XPath functions (`document`, `format-number`, `current`, ...)
- stylesheet `import`/`include`
- built-in template rules

+ some technical details.

Links to more information

www.w3.org/Style/XSL/

W3C's XSL homepage, contains lots of links

www.w3.org/TR/xslt

The XSLT specification (a W3C Recommendation)

www.mulberrytech.com/xsl/xsl-list/

XSL-List - mailing list

metalab.unc.edu/xml/books/bible/updates/14.html

a chapter from "The XML Bible" on XSL Transformations

nwalsh.com/docs/tutorials/xml99

an XSL tutorial by Paul Grosso and Norman Walsh

www.alphaworks.ibm.com/tech/LotusXSL

LotusXSL, a Java XSLT implementation from IBM alphaWorks

users.iclway.co.uk/mhkay/saxon/

SAXON, another Java implementation

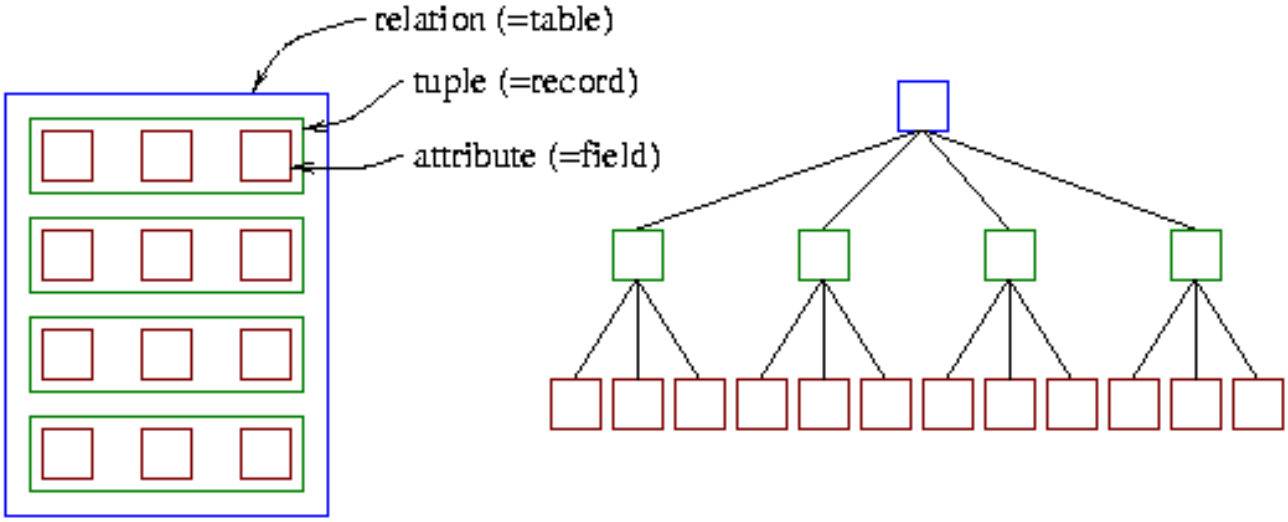
www.jclark.com/xml/xt.html

XT (by James Clark, the editor of the XSLT spec), yet another Java implementation

XML-QL - an XML query language

Queries on XML documents

XML documents generalize relational data.



A relation is a tree of height two with

- *unbounded* fanout at the first level;
- *fixed* fanout at the second level.

In contrast, an XML document is an arbitrary tree.

How should query languages like SQL be similarly generalized?

Usage scenarios

XML querying is relevant for:

human-readable documents

to retrieve individual documents, to provide dynamic indexes, to perform context-sensitive searching, and to generate new documents.

data-oriented documents

to query (virtual) XML representations of databases, to transform data into new XML representations, and to integrate data from multiple heterogeneous data sources.

mixed-model documents

to perform queries on documents with embedded data, such as catalogs, patient health records, employment records, or business analysis documents.

Query language requirements

The W3C Query Working Group has identified many technical requirements:

- at least one **XML syntax**; at least one human-readable syntax.
- must be **declarative**;
- must be protocol independent;
- must respect XML data model;
- must be **namespace aware**;
- must **coordinate with XML Schema**;
- must work even if schemas are unavailable;
- must support simple and complex datatypes;
- must support **universal and existential quantifiers**;
- must support operations on hierarchy and sequence of document structures;
- must **combine information from multiple documents**;
- must support aggregation;
- must be **able to transform and to create XML structures**;
- must be able to traverse ID references.

In short, it must be SQL generalized to XML!

Early query language proposals

A doc-head proposal:

- XQL

Three database proposals:

- XML-QL
- YATL
- Lorel

They are all quite similar, but XML-QL has the nicest look-and-feel.

The XML-QL language

The XML-QL language is designed with the following features:

- it is **declarative**, like SQL.
- it is **relational complete**, e.g. it can express joins.
- it can be implemented with known database techniques.
- it can **extract** data from exisiting XML documents *and* **construct** new XML documents.

XML-QL is implemented as a prototype and is freely available in a Java version.

The XML-QL data model

XML-QL works on an **abstraction**, called an XML graph, of the concrete XML document:

- **comments** and **processing instructions** are ignored;
- the relative **order** of elements is ignored;
- every node has an **ID** (autogenerated, if necessary);
- all leaves are character data.

XML graphs are obtained from XML documents but are also generated by queries.

A graph is mapped back into an XML document by choosing arbitrary orderings of element sequences.

This abstraction is very similar to that from tables to relations: *disregard the order of tuples and attributes*.

Matching data using patterns

A simple query to select those authors that have published for Addison-Wesley:

```
WHERE <bib><book>
        <publisher><name>"Addison-Wesley"</name></publisher>
        <title>  $t </title>
        <author> $a </author>
      </book></bib> IN "bib.xml"
CONSTRUCT $a
```

- the `$t` and `$a` are variables that pick out contents.
- the output is a collection of author names.

A more convenient syntax is allowed:

```
WHERE <bib><book>
        <publisher><name>"Addison-Wesley"</></>
        <title>  $t </>
        <author> $a </>
      </></> IN "bib.xml"
CONSTRUCT $a
```

Note that XML-QL is not in XML syntax...

Constructing XML data

Results of a query can be wrapped in XML:

```
WHERE <bib><book>
        <publisher> <name>"Addison-Wesley" </> </>
        <title>  $t </>
        <author> $a </>
      </></> IN "bib.xml"
CONSTRUCT <result>
             <author> $a </>
             <title>  $t </>
          </>
```

- results are grouped in elements.
- the pattern matches once for each author, which may give duplicates of books.

The following example data:

```
<bib>
  <book year="1995">
     <!-- A good introductory text -->
     <title>An Introduction to Database Systems </title>
     <author><lastname>Date </lastname></author>
     <publisher><name>Addison-Wesley </name> </publisher>
  </book>
  <book year="1998">
     <title>Foundation for Object/Relational Databases</title>
     <author><lastname>Date </lastname></author>
     <author><lastname>Darwen </lastname></author>
     <publisher><name>Addison-Wesley </name>  </publisher>
  </book>
</bib>
```

produces the output:

```
<XML>
 <result>
    <author><lastname>Date </lastname></author>
    <title>An Introduction to Database Systems </title>
 </result>
 <result>
    <author><lastname>Date </lastname></author>
    <title>Foundation for Object/Relational Databases</title>
 </result>
 <result>
    <author><lastname>Darwen </lastname></author>
    <title>Foundation for Object/Relational Databases</title>
 </result>
</XML>
```

Creating an explicit root element

Every XML document must have a single root.

XML-QL supplies an `<XML>` element as default, but others may be specified:

```
CONSTRUCT <results> {
 WHERE <bib><book>
         <publisher> <name>"Addison-Wesley" </> </>
         <title>  $t </>
         <author> $a </>
       </></> IN "bib.xml"
 CONSTRUCT <result>
             <author> $a </>
             <title>  $t </>
           </>
} </results>
```

Grouping with nested queries

Queries can be nested arbitrarily:

```
WHERE <book> $p </> IN "bib.xml",

      <title> $t </>
      <publisher> <name> "Addison-Wesley" </> </> IN $p
CONSTRUCT <result>
             <title> $t </>
             { WHERE  <author> $a </> IN $p
               CONSTRUCT <author> $a </>
             }
          </>
```

- `$p` matches the contents of each book.
- `$t` matches titles of Addison-Wesley books.
- each title is included once in the reault.
- in the nested query, `$a` matches all authors of the given title.

This will group authors of the same book:

```
<XML>
 <result>
    <title> An Introduction to Database Systems </title>
    <author> <lastname> Date </lastname> </author>
 </result>

 <result>
    <title> Foundation for Object/Relational Databases</title>
    <author> <lastname> Date </lastname> </author>
    <author> <lastname> Darwen </lastname> </author>
 </result>
</XML>
```

Joining elements by value

Joins are expressed by using multiple queries that share variables:

```
WHERE <bib.article>
        <author>
          <firstname.PCDATA> $f </>  // firstname $f
          <lastname.PCDATA>  $l </>  // lastname $l
        </>
      </> CONTENT_AS $a IN "bib.xml",

      <book year=$y>
        <author>
          <firstname.PCDATA>$f</> // join on same firstname $f
          <lastname.PCDATA>$l</>  // join on same lastname $l
        </>
      </> IN "bib.xml",

      $y > 1995
CONSTRUCT <article> $a </>
```

- `$f` and `$l` join pairs in the two documents.
- `$y` is used to constrain the year.
- the `CONTENT_AS` saves us from reconstructing an entire fragment.
- all chardata nodes are implictly wrapped into a `PCDATA` element.
- ('`.`' in element names is explained later...)

This retrieves all articles that have at least one author who has also written a book since 1995.

Tag variables

Element names can also be bound by variables:

```
WHERE <$p>
        <title> $t </title>
        <year> 1995 </>
        <$e> Smith </>
      </> IN "www.a.b.c/bib.xml",
      $e IN {author, editor}
CONSTRUCT <$p>
            <title> $t </title>
            <$e> Smith </>
          </>
```

- `$p` matches `book` and `article`.
- `$e` matches `author` and `editor`.
- this saves us from writing four queries.

This finds all publications in 1995 where Smith is either author or editor.

Regular-path expressions

The context of an element can be expressed in a CSS- and DSD-like manner:

```
WHERE <part*> <name> $r </> <brand> Ford </> </> IN "www.a.b.c/parts.xml"
CONSTRUCT <result> $r </>
```

However, arbitrary regular expressions are supported:

```
WHERE <part+.(subpart|component.piece)>$r</> IN "www.a.b.c/parts.xml"
CONSTRUCT <result> $r</>
```

Merging results with Skolem functions

ID values can be used to guide the grouping of data:

```
WHERE <$> <author> <firstname> $fn </>
                   <lastname> $ln </>
          </>
          <title> $t </>
      </> IN "www.a.b.c/bib.xml"
CONSTRUCT <person ID=PersonID($fn, $ln)>
              <firstname ID=FirstnameID($fn, $ln)> $fn </>
              <lastname ID=LastnameID($fn, $ln)> $ln </>
              <publicationtitle> $t </>
          </>
```

- `PersonID($fn,$ln)` creates a unique ID for each combination.
- the `PersonID` identifier creates a "namespace".
- in the result, elements with the same ID are *merged*.
- merging constructs the union of content elements (duplicates are removed).

All `person` elements with the same ID have their contents merged. (In general, this corresponds to unbounded nesting of queries.)

Integrating data from multiple XML sources

Queries can combine information from many sources:

```
WHERE <person>
        <name></> ELEMENT_AS $n
        <ssn> $ssn </>
      </> IN "www.a.b.c/data.xml",

      <taxpayer>
        <ssn> $ssn </>
        <income></> ELEMENT_AS $i
      </> IN "www.irs.gov/taxpayers.xml"
CONSTRUCT <result> <ssn> $ssn </>  $n $i </>
```

Recovering ordinary relations

A standard relation with schema `(name,age,email)` can be represented as an XML document:

```
<relation>
  <tuple>
    <name>John Doe</name>
    <age>38</age>
    <email>john.doe@widget.com</email>
  </tuple>
  <tuple>
    <name>Joe Blow</name>
    <age>42</age>
    <email>joe.blow@gadget.com</email>
  </tuple>
</relation>
```

Fun exercise: express relational algebra in XML-QL through this encoding.

The query language of the future

The official W3C XML Query language will:

- share many similarities with XML-QL;
- have a different syntax (in XML);
- interface with relational databases;
- use native query engines (Oracle, DB2, ...);
- be presented in 2001.

Members of the W3C Query Working Group have proposed Quilt as the query language of the future. It is based on XPath.

Links to more information

www.w3.org/TR/NOTE-xml-ql

the XML-QL W3C Note

www.research.att.com/~mff/xmlql/doc

the XML-QL home page

www.w3.org/XML/Activity.html#query-wg

the XML Query Working Group

www.w3.org/TR/xmlquery-req

XML Query Requirements (W3C Working Draft)

www.oasis-open.org/cover/xmlQuery.html

Robin Cover's page on XML query languages

www.almaden.ibm.com/cs/people/chamberlin/quilt.html

The Quilt language.

Background: W3C

1. W3C - The World Wide Web Consortium
2. Policies
3. Organization
4. Activities
5. Events
6. Technical Reports

W3C - The World Wide Web Consortium (www.w3.org)

- the de facto leader in defining Web standards

Consists of almost 400 companies and organizations, led by Tim Berners-Lee, creator of the World Wide Web.

W3C's Mission Statement:

"To lead the evolution of the Web - the universe of information accessible through networked computers."

Coming up: an overview of the W3C Process Document...

Policies

- consensus - reach substantial agreement
- dissemination - limit "intellectual property rights"

(+the unofficial: better too soon than too late)

Organization

- the Members (who pay $50000 a year) - companies and organizations (carry out activities)
- the Team - Chairman, Director, Staff (coordinate activities, hosted by MIT, INRIA, and KEIO)
- the Offices - regional organizations (promote activities)
- Advisory Commitee (member representatives) - reviews proposals etc.
- Advisory Board (elected) - provides guidance of strategy, conflict resolution, etc.
- Communication Team - responsible for information dissemination

Activities

- areas of Web technologies or policies

Carried out by groups:

- Working Groups - produce specifications and prototypes
- Interest Groups - explore and evaluate technologies
- Coordination Groups - ensure consistency and integrety between other groups

Current XML groups: query, schema, linking, core, coordination
Other activities: HTML, HTTP, P3P, Amaya, ...

Events

Organization of events:

- workshops - short expert meeting
- symposia - education
- conferences - the International World Wide Web Conference

Technical Reports

- the central activity of W3C

Member submissions and Recommendation track:

- Notes - acknowledged submissions by Members, Working Group notes, etc.
- Working Drafts - Working Group reports (work in progress)
- Candidate Recommendations - stable Working Drafts
- Proposed Recommendations - being reviewed by the Advisory Committee
- Recommendations - standards recommended by W3C

Submission of Notes:

1. submission (Members only!)
2. acknowledgement/rejection by Director

Recent BRICS Notes Series Publications

NS-00-8 Anders Møller and Michael I. Schwartzbach. *The XML Revolution*. December 2000. 149 pp.

NS-00-7 Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. *Document Structure Description 1.0*. December 2000. 40 pp.

NS-00-6 Peter D. Mosses and Hermano Perrelli de Moura, editors. *Proceedings of the Third International Workshop on Action Semantics, AS 2000*, (Recife, Brazil, May 15–16, 2000), August 2000. viii+148 pp.

NS-00-5 Claus Brabrand. *`<bigwig>` Version 1.3 — Tutorial*. September 2000. ii+92 pp.

NS-00-4 Claus Brabrand. *`<bigwig>` Version 1.3 — Reference Manual*. September 2000. ii+56 pp.

NS-00-3 Patrick Cousot, Eric Goubault, Jeremy Gunawardena, Maurice Herlihy, Martin Raussen, and Vladimiro Sassone, editors. *Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency Theory, GETCO '00*, (State College, USA, August 21, 2000), August 2000. vi+116 pp.

NS-00-2 Luca Aceto and Björn Victor, editors. *Preliminary Proceedings of the 7th International Workshop on Expressiveness in Concurrency, EXPRESS '00*, (State College, USA, August 21, 2000), August 2000. vi+130 pp.

NS-00-1 Bernd Gärtner. *Randomization and Abstraction — Useful Tools for Optimization*. February 2000. 106 pp.

NS-99-3 Peter D. Mosses and David A. Watt, editors. *Proceedings of the Second International Workshop on Action Semantics, AS '99*, (Amsterdam, The Netherlands, March 21, 1999), May 1999. iv+172 pp.

NS-99-2 Hans Hüttel, Josva Kleist, Uwe Nestmann, and António Ravara, editors. *Proceedings of the Workshop on Semantics of Objects As Processes, SOAP '99*, (Lisbon, Portugal, June 15, 1999), May 1999. iv+64 pp.

NS-99-1 Olivier Danvy, editor. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '99*, (San Antonio, Texas, USA, January 22–23, 1999), January 1999.