Outline

Challenges of Electronic Business

Specification Approaches

Commitments

Architecture in IT

Contracts and Governance

XML Concepts and Techniques

XML Modeling and Storage

Summary and Directions
Outline

Challenges of Electronic Business

Specification Approaches

Commitments

Architecture in IT

Contracts and Governance

XML Concepts and Techniques

XML Representation

XML Query and Manipulation

XPath

XQuery

XML Modeling and Storage
XML Representation

- Concepts
- Parsing and Validation
- Schemas
What is Metadata?

Literally, data about data

▶ Description of data that captures some useful property regarding its
  ▶ Structure and meaning
  ▶ Provenance: origins
  ▶ Treatment as permitted or allowed: storage, representation, processing, presentation, or sharing

▶ Markup is metadata pertaining to media artifacts (documents, images), generally specified for suitable parsable units
Motivations for Metadata

Mediating information structure (surrogate for meaning) over time and space

- Storage: extend life of information
- Interoperation for business
- Interoperation (and storage) for regulatory reasons: supporting organizational coherence

General themes

- Make meaning of information (more) “explicit”
- Enable reuse across applications: repurposing (compare to screen-scraping)
- Enable better tools to improve productivity

Reduce need for detailed prior agreements
Metadata History

What kind and how much of prior agreement do you need?

- No markup: significant prior agreement
- CSV, Comma (likewise Tab) Separated Values: no nesting
- Ad hoc tags
- SGML (Standard Generalized Markup L): complex, few reliable tools; used for document management
- HTML (HyperText ML): simplistic, fixed, unprincipled vocabulary that mixes structure and display
- XML (eXtensible ML): simple, yet extensible subset of SGML to capture custom vocabularies
  - Machine processible
  - Comprehensible to people: easier debugging
Uses of XML

Supporting arms-length relationships

- Exchanging information across software components, even within an administrative domain
- Storing information in nonproprietary format
- Representing semistructured descriptions:
  - Products, services, catalogs
  - Contracts
  - Queries, requests, invocations, responses (as in SOAP): basis for Web services
  - System configurations
Example XML Document

```xml
<?xml version="1.0"?> <!-- processing instruction -->
<topelem attr0="foo"> <!-- exactly one root -->
  <subelem attr1="v1" attr2="v2">
    Optional text (PCDATA) <!-- parsed character data -->
    <subsubelem attr1="v1" attr2="v2"/>
  </subelem>
  <null_elem/>
  <short_elem attr3="v3"/>
</topelem>
```
Exercise

Produce an example XML document corresponding to a directed graph
Compare with Lisp

List processing language

- S-expressions
- Cons pairs: `car` and `cdr`
- Lists as nil-terminated s-expressions
- Arbitrary structures built from few primitives
- Untyped
- Easy parsing
- Regularity of structure encourages recursion
Exercise

Produce an example XML document corresponding to

- An invoice from Locke Brothers for 100 units of door locks at $19.95, each ordered on 15 January and delivered to Custom Home Builders
- Factor in certified delivery via UPS for $200.00 on 18 January
- Factor in addresses and contact info for each party
- Factor in late payments
Meaning in XML

- Relational DBMSs work for highly structured information, but rely on column names for meaning.
- Same problem in XML (reliance on names for meaning) but better connections to richer meaning representations.
  - Leads to a need for a richer way of specifying a *vocabulary*, i.e., such names suitably organized.
XML Namespaces: 1

- Because XML supports custom vocabularies and interoperation, there is a high risk of name collision
- A namespace is a collection of names
- Namespaces must be identical or disjoint
  - Crucial to support independent development of vocabularies
  - Rely upon and provide a naming convention
  - Examples
    - MAC addresses
    - Postal and telephone codes
    - Vehicle identification numbers
    - IP addresses and domains as for the Internet
    - On the Web, use URIs for uniqueness
XML Namespaces: 2

Qualified names

```xml
<!−− xml* is reserved −−>
<?xml version="1.0"?>
<arbıt:top xmlns="a URI" <!−− default namespace −−>
    xmlns:arbıt="http://wherever.it.might.be/arbıt−ns"
    xmlns:random="http://another.one/random−ns">
<arbıt:aElem attr1="v1" attr2="v2">
    Optional text (PCDATA)
    <arbıt:bElem attr1="v1" attr2="v2"/>
</arbıt:aElem>
<random:simple_elem/>
<random:aElem attr3="v3"/>
<!−− compare arbıt:aElem −−>
</arbıt:top>
```
Uniform Resource Identifier
Key abstraction underlying Web architecture

- URIs are abstract
- What matters is their (purported) uniqueness
- URIs have no proper syntax per se
- Kinds of URIs
  - URLs, as in browsing: not used in standards any more
    - Formal syntax
    - A way to resolve to a resource
  - URNs, which leave the mapping of names to locations up in the air
    - Formal syntax
- Good design: the URI resource exists
  - Ideally, as a description of the resource in RDDL
  - Use a URL or URN
RDDL

Resource Directory Description Language

Not a formal standard

- A way to provide (human readable) content for a namespace URI
  - No technical bearing of such content, since a URI is merely an identifier
- Captures namespace description for people
  - XML Schema
  - Text description
Well-Formedness and Parsing

If it isn’t well-formedness, it isn’t XML

- An XML document maps to a parse tree, not a forest
  - Each element must end (exactly once): obvious nesting structure (one root)
  - An attribute can have at most one occurrence within an element; an attribute’s value must be a quoted string
- Well-formed XML documents can be parsed
XML InfoSet

A standardization of the low-level aspects of XML

- What an element looks like
- What an attribute looks like
- What comments and namespace references look like
- Ordering of attributes is irrelevant
- Representations of strings and characters

Primarily directed at tool vendors to ensure round-tripping
Elements Versus Attributes: 1

- Elements are essential for constructing an XML tree: structure and expressiveness
  - Have subelements and attributes
  - Can be repeated
  - Loosely might correspond to independently existing entities or associations
  - Can capture all there is to attributes
Elements Versus Attributes: 2

- Attributes are not essential
  - End of the road: no subelements or subattributes
  - Like text; restricted to string values
  - Guaranteed unique for each element
  - Capture adjunct information about an element
  - Great as references to elements

Good idea to use in such cases to improve readability
Elements Versus Attributes: 3

\[
\begin{align*}
\text{<invoice>} & \\
\text{<price currency='USD'>} & \\
\phantom{<price currency='USD'>} & 19.95 \\
\text{</price>} & \\
\text{</invoice>} & \\
\text{Or} & \\
\text{<invoice amount='19.95' currency='USD'/>} & \\
\text{Or even} & \\
\text{<invoice amount='USD 19.95'/>} & 
\end{align*}
\]
Outline

Challenges of Electronic Business

Specification Approaches

Commitments

Architecture in IT

Contracts and Governance

XML Concepts and Techniques
  XML Representation
  XML Query and Manipulation
    XPath
    XQuery

XML Modeling and Storage
Main XML query and manipulation languages include

- XPath
- XQuery
- XSLT
- SQL/XML
Metaphors for Handling XML: 1

How we conceptualize XML documents determines our approach for handling them

- **Text**: an XML document is text
  - Ignore any structure and perform simple pattern matches
- **Tags**: an XML document is text interspersed with tags
  - Treat each tag as an “event” during reading a document and specify callbacks, as in SAX (Simple API for XML)
  - Construct regular expressions as in screen scraping

*Abolish the word “tag” from your vocabulary*
Metaphors for Handling XML: 2

- **Tree**: an XML document is a tree
  - Walk the tree using DOM (Document Object Model)
- **Template**: an XML document has regular structure
  - Let XPath, XSLT, XQuery do the work
- **Thought**: an XML document represents an information model
  - Access knowledge via RDF or OWL
XPath

Used as part of XPointer, SQL/XML, XQuery, and XSLT

- Models XML documents as trees with nodes
  - Elements
  - Attributes
  - Text (PCDATA)
  - Comments
  - Root node: above root of document
  - Namespace declarations
  - Processing instructions
Achtung!

- Parent in XPath is like parent as traditionally in computer science
- Child in XPath is confusing:
  - An attribute is not a child of its parent
  -Makes a difference for recursion (e.g., in XSLT `apply-templates`)
- Our terminology follows computer science:
  - e-children, a-children, t-children
  - Sets via et-, ta-, and so on
XPath Location Paths: 1
Ways to walk a tree

- Relative or absolute
- Reminiscent of file system paths, but *much* more subtle
  - Name of an element to walk down
  - Leading `/`: root
  - `/`: indicates walking down a tree
  - `.:` currently matched (*context*) node
  - `..`: parent node
XPath Location Paths: 2

- `@attr`: to check existence or access value of the given attribute
- `text()`: extract all text nodes (within the context node)
- `comment()`: extract all comments
- `[ ]`: generalized array accessors
- Variety of axes, discussed below
XPath Navigation

- Select children according to position, e.g., [j], where j could be 1 ... last()
- Descendant-or-self operator, //
  - ./elem finds all elems under the current node
  - //elem finds all elems in the document
- Wildcard, *:
  - collects e-children (subelements) of the node where it is applied, but omits the t-children
  - @*: finds all attribute values
XPath Queries (Selection Conditions)

- Attributes: //Song[@genre="jazz"]
- Text (implicit `text()`): //Song[starts-with(.//group, "Led")]
- Existence of attribute: //Song[@genre]
- Existence of subelement: //Song[group]
- Boolean operators: and, not, or
- Set operator: union (|), analogous to choice
- Arithmetic operators: >, <, ...
- String functions: contains(), concat(), length(), starts-with(), ends-with()
- `distinct-values()`
- Aggregates: `sum()`, `count()`
XPath Axes: 1

Axes are addressable node sets based on the document tree and the current node

- Axes facilitate navigation of a tree
- Several are defined
- Mostly straightforward but some of them order the nodes as the reverse of others
- Some captured via special notation
  - current, child, parent, attribute, ...
XPath Axes: 2

- **preceding**: nodes that end before the start of the context node (not ancestors, attributes, namespace nodes)
- **following**: nodes that start after the end of the context node (not descendants, attributes, namespace nodes)
- **preceding-sibling**: preceding nodes that are children of the same parent, in reverse document order
- **following-sibling**: following nodes that are children of the same parent
XPath Axes: 3

- **ancestor**: *proper* ancestors, i.e., element nodes (other than the context node) that contain the context node, in reverse document order
- **descendant**: proper descendants
- **ancestor-or-self**: ancestors, including self (if it matches the next condition)
- **descendant-or-self**: descendants, including self (if it matches the next condition)
XPath Axes: 4

- Longer syntax: `child::Song`
- Some captured via special notation
  - `self::*`:
  - `child::node(): node()` matches all nodes that are children of the context node
  - `preceding::*`
  - `descendant::text()`
  - `ancestor::Song`
  - `descendant-or-self::node()`, which abbreviates to `//`
  - Compare `/descendant-or-self::Song[1]` (first descendant Song) and `//Song[1]` (first Songs (children of their parents))
XPath Axes: 5

► Each axis has a principal node kind
  ► attribute: attribute
  ► namespace: namespace
  ► All other axes: element

► * matches whatever is the principal node kind of the current axis

► node() matches all nodes
XPointer

Enables pointing to specific parts of documents

- Combines XPath with URLs
- URL to get to a document; XPath to walk down the document
- Can be used to formulate queries, e.g.,
  - Song-URL#xpointer(//Song[@genre="jazz"])
  - The part after # is a fragment identifier

- Fine-grained addressability enhances the Web architecture

High-level “conceptual” identification of node sets
XQuery

- The official query language for XML, now a W3C recommendation, as version 1.0
- Given a non-XML syntax, easier on the human eye than XML
- An XML rendition, XqueryX, is in the works
XQuery Basic Paradigm

The basic paradigm mimics the SQL (SELECT–FROM–WHERE) clause for $x \text{ in } \text{doc}('q2.xml')//Song where $x/@lg = 'en'
return
$\langle \text{English-Sgr name=}'{$x/Sgr/@name}' \text{ ti=}'{$x/@ti}'\rangle$
FLWOR Expressions

Pronounced “flower”

- At least one of these:
  - For: iterative binding of variables over range of values
  - Let: one shot binding of variables over vector of values

- Zero or one of these:
  - Where
  - Order by (sort: optional)

- Exactly one of this:
  - Return
The **for** clause

- Introduces one or more variables
  - Analogous to what we might term a “tuple” variable, whose each possible value is a tuple
- Generates possible bindings for each variable
- Acts as a mapping functor or iterator
  - In essence, all possible combinations of bindings are generated: like a Cartesian product in relational algebra
  - The bindings form an ordered list
XQuery Where Clause

The **where** clause

- Selects the combinations of bindings that are desired
- Behaves like the **where** clause in SQL, in essence producing a join based on the Cartesian product
XQuery Return Clause

The **return** clause

- Specifies what node-sets are returned based on the selected combinations of bindings
  - Constructs the results
  - Includes support for macro-like features
XQuery Let Clause

The **let** clause
- Like **for**, introduces one or more variables
  - Analogous to what we might term a “relation” variable, whose each possible value is a sequence of tuples
- Like **for**, generates possible bindings for each variable
- Unlike **for**, generates the bindings as a list in one shot (no iteration)
XQuery Order By Clause

The **order by** clause

- Specifies how the vector of variable bindings is to be sorted before the return clause
- Sorting expressions can be nested by separating them with commas
- Variants allow specifying
  - `descending` or `ascending` (default)
  - `empty greatest` or `empty least` to accommodate empty elements
  - stable sorts: `stable order by`
  - collations: `order by $t collation` collation-URI: (obscure, so skip)
XQuery Positional Variables

The **for** clause can be enhanced with a positional variable

- A positional variable captures the position of the main variable in the given **for** clause with respect to the expression from which the main variable is generated

- Introduce a positional variable via the **at** $var$ construct
The **declare** clause specifies things like

- **Namespaces:** declare namespace pref='value'
  - Predefined prefixes include XML, XML Schema, XML Schema-Instance, XPath, and **local**

- **Settings:** declare boundary-space preserve (or strip)

- **Default collation:** a URI to be used for collation when no collation is specified
XQuery Quantification: 1

- Two quantifiers **some** and **every**
- Each quantifier expression evaluates to true or false
- Each quantifier introduces a bound variable, analogous to **for**

```xml
for $x in ...
where some $y in ...
satisfies $y ... $x
return ...
```

Here the second $x refers to the **same** variable as the first
A typical useful quantified expression would use variables that were introduced outside of its scope

- The order of evaluation is implementation-dependent: enables optimization
- If some bindings produce errors, this can matter
- **some**: trivially false if no variable bindings are found that satisfy it
- **every**: trivially true if no variable bindings are found
for, let, some, and every introduce variables

- The visibility of a variable follows typical scoping rules
- A variable referenced within a scope is
  - Bound if it is declared within the scope
  - Free if it not declared within the scope

```xml
for $x$ in ...
where some $x$ in ...
satisfies ...
return ...
```

Here the two $x$ refer to different variables
XQuery Conditionals

Like a classical **if-then-else** clause

- The **else** is not optional
- Empty sequences or node sets, written ( ), indicate that nothing is returned
XQuery Constructors

Braces \{ \} to delimit expressions that are evaluated to generate the content to be included; analogous to macros

- document \{ \}: to create a document node with the specified contents
- element \{ \} \{ \}: to create an element
  - element foo \{ 'Bar' \}: creates \textlt{\textlt{foo}Bar\textlt{/foo}}
  - element \{ 'foo' \} \{ 'Bar' \}: evaluates the name expression
- attribute \{ \} \{ \}: likewise
- text \{ body\}: simpler, because anonymous
XQuery Effective Boolean Value

Analogous to Lisp, a general value can be treated as if it were a Boolean

- A `xs:boolean` value maps to itself
- An empty sequence maps to `false`
- A sequence whose first member is a node maps to `true`
- A numeric that is 0 or NaN maps to `false`, else to `true`
- An empty string maps to `false`, others to `true`
Defining Functions

```xml
declare function local:itemftop($t)
{
  local:itemf($t,())
};
```

- Here `local:` is the namespace of the query
- The arguments are specified in parentheses
- All of XQuery may be used within the defining braces
- Such functions can be used in place of XPath expressions
Functions with Types

declare function local:itemftop($t as element())
as element() *
{
    local:itemf($t,())
}

- Return types as above
- Also possible for parameters, but ignore such for this course