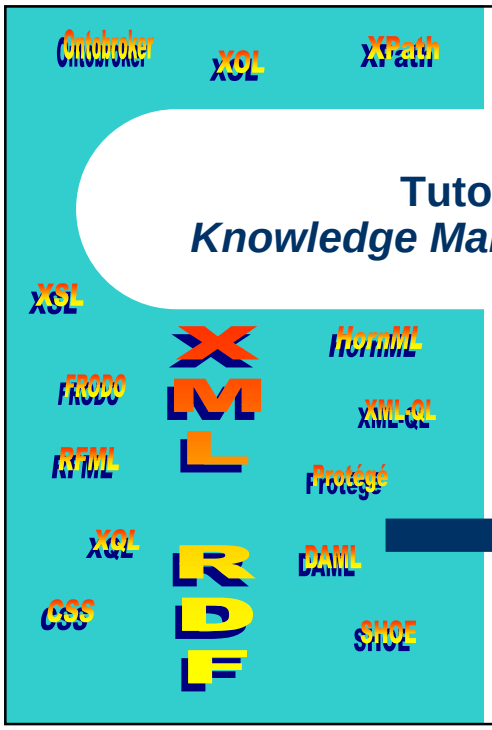


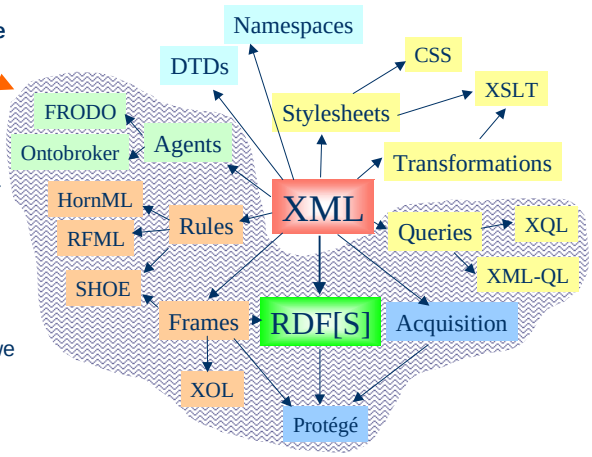
Tutorial on *Knowledge Markup Techniques*

Harold Boley
Stefan Decker
Michael Sintek

ECAI 2000 Berlin
22 August 2000

Overview and Tutorial Mindmap

- Increasing demand for **formalized knowledge** on the Web: **AI's** chance!
- XML-based markup languages provide a 'universal' **storage** and **interchange** format for such **Web-distributed knowledge representation**
- Tutorial introduces key techniques for **knowledge markup**: we show how to marry AI representations (e.g., logics and frames) and XML (incl. RDF and RDF Schema)



Extensible Markup Language

XML

General Advantages of XML for KR

XML offers new general possibilities, from which AI knowledge representation (KR) can profit:

(1) Definition of self-describing data in worldwide standardized, non-proprietary format

(2) Structured data and knowledge exchange for enterprises in various industries

(3) Integration of information from different sources (into uniform documents)

XML

Specific Advantages of XML for KR

XML provides the most suitable infrastructure for knowledge bases on the Web (incl. for W3C languages such as RDF)

Additional special KR uses of XML are:

- Uniform storage of knowledge bases
- Interchange of knowledge bases between different AI languages
- Exchange between knowledge bases and databases, application systems, etc.

Even transformation/compilation of AI source programs using XML markup and annotations is possible

Address Example: External to HTML

External Presentation:

Xaver M. Linde
Wikingerufer 7
10555 Berlin

HTML tags are still presentation-oriented

HTML Markup:

```
<em>Xaver M. Linde</em>
<br>
Wikingerufer 7
<br>
<strong>10555 Berlin</strong>
```

Address Example: HTML to XML

HTML Markup:

```
<em>Xaver M. Linde</em>
<br>
Wikingerufer 7
<br>
<strong>10555 Berlin</strong>
```

XML tags are chosen for representation needs

XML Markup:

```
<address>
  <name>Xaver M. Linde</name>
  <street>Wikingerufer 7</street>
  <town>10555 Berlin</town>
</address>
```


Address Example: XML to External

XML Markup:

```
<address>
  <name>Xaver M. Linde</name>
  <street>Wikingerufer 7</street>
  <town>10555 Berlin</town>
</address>
```

XML stylesheets are, e.g., usable to generate different presentations

External Presentations:

Xaver M. Linde
Wikingerufer 7
10555 Berlin

. . .

Xaver **M**. **L**inde
Wikingerufer 7
10555 Berlin

Address Example: XML to XML

XML Markup 1:

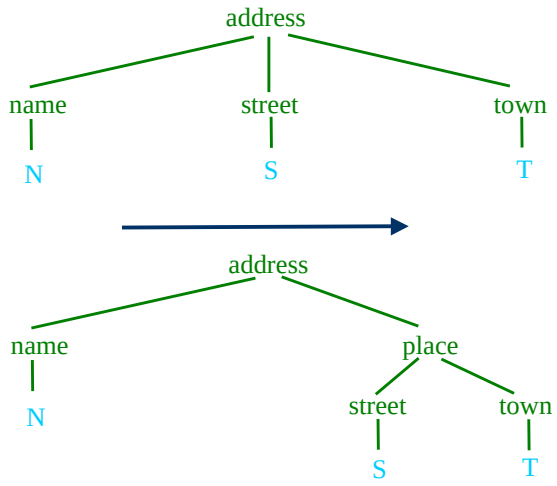
```
<address>
  <name>Xaver M. Linde</name>
  <street>Wikingerufer 7</street>
  <town>10555 Berlin</town>
</address>
```

XML Markup 2:

```
<address>
  <name>Xaver M. Linde</name>
  <place>
    <street>Wikingerufer 7</street>
    <town>10555 Berlin</town>
  </place>
</address>
```

XML stylesheets are also usable to transform XML representations

Address Example: Some Stylesheets Will Contain Term-(Tree-)Rewriting Rules



Address Example: XML Queries

XML Markup:

element

```
<address>          subelements
  <name>Xaver M. Linde</name>
  <street>Wikingerufer 7</street>
  <town>10555 Berlin</town>
</address>
```

XML Query (XML-QL):

```
WHERE
  <address>
    <name>Xaver M. Linde</name>
    <street>$s</street>
    <town>$t</town>
  </address>
CONSTRUCT
  <binding>
    <s>$s</s>
    <t>$t</t>
  </binding>
```

XML queries can select subelements of XML elements

```
<binding>
  <s>Wikingerufer 7</s>
  <t>10555 Berlin</t>
</binding>
```


Address Example: Prolog Queries

Prolog Term:

structure

```
address(    substructures
  name("Xaver M. Linde"),
  street("Wikingerufer 7"),
  town("10555 Berlin")
)
```

Prolog Query:

```
address(
  name("Xaver M. Linde"),
  street(S),
  town(T)
)
```

Prolog queries can select substructures of Prolog structures

```
S = "Wikingerufer 7"
T = "10555 Berlin"
```

Address Example: The Element Tree

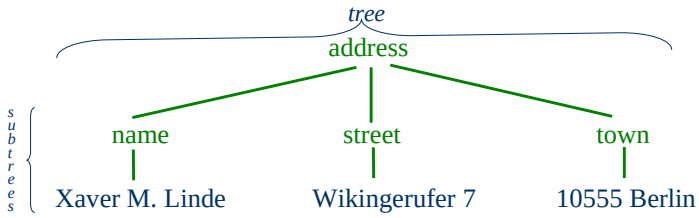
XML Markup:

element:
```
<address>
  <name>Xaver M. Linde</name>
  <street>Wikingerufer 7</street>
  <town>10555 Berlin</town>
</address>
```
subelements

Prolog Term:

structure:
```
address(
  name("Xaver M. Linde"),
  street("Wikingerufer 7"),
  town("10555 Berlin")
)
```
substructures

Node-Labeled, (Left-to-Right-)Ordered Element Tree:



Address Example: Document Type Definition and Tree (1)

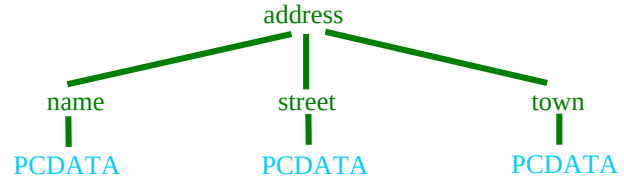
Document Type Definition (DTD):

```
<!ELEMENT address (name, street, town) >
<!ELEMENT name    (#PCDATA) >
<!ELEMENT street  (#PCDATA) >
<!ELEMENT town    (#PCDATA) >
```

Extended Backus-Naur Form (EBNF):

```
address ::= name street town
name    ::= PCDATA
street  ::= PCDATA
town    ::= PCDATA
```

Document Type Tree:

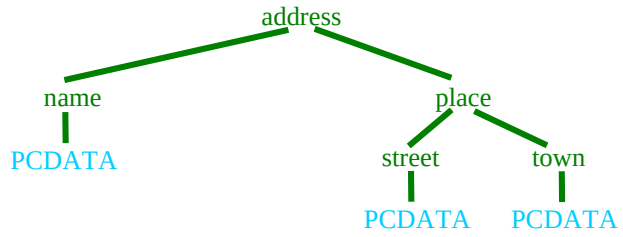


Address Example: Document Type Definition and Tree (2)

Document Type Definition (DTD):

```
<!ELEMENT address   (name, place) >
<!ELEMENT place     (street, town) >
<!ELEMENT name      (#PCDATA) >
<!ELEMENT street    (#PCDATA) >
<!ELEMENT town      (#PCDATA) >
```

Document Type Tree:



Well-Formedness and Validity

XML principles for a document being *well-formed*:

- Open and close all tags
- Empty tags end with />
- There is a unique root element
- Elements may not overlap
- Attribute values are quoted
- < and & are only used to start tags and entities
- Only the five predefined entity references are used

XML principle for a document being *valid with respect to (w.r.t.) a DTD* :

- Match the constraints listed in the DTD (or, generate from DTD as linearized derivation tree, as shown later)

Checked by **validators** such as http://www.stg.brown.edu/service/xmlvalid/

Mail-Box Example: Address Variant

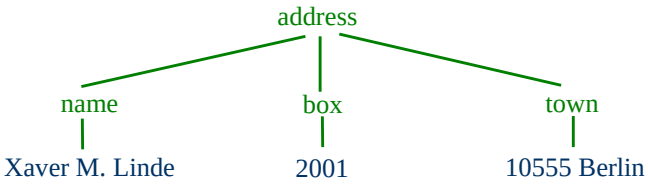
XML Markup:

```
<address>
  <name>Xaver M. Linde</name>
  <box>2001</box>
  <town>10555 Berlin</town>
</address>
```

Prolog Term:

```
address(
  name("Xaver M. Linde"),
  box("2001"),
  town("10555 Berlin")
)
```

Node-Labeled, (Left-to-Right-)Ordered Element Tree:



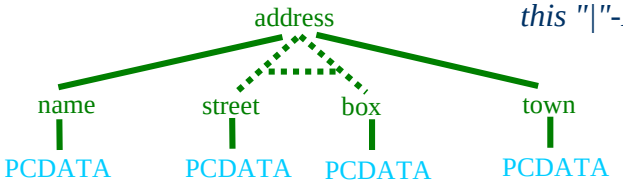
"|"-Disjoined Street/Mail-Box Example: Document Type Definition and Tree

Document Type Definition (DTD):

```
<!ELEMENT address  (name, (street | box), town) >
<!ELEMENT name     (#PCDATA) >
<!ELEMENT street   (#PCDATA) >
<!ELEMENT box      (#PCDATA) >
<!ELEMENT town     (#PCDATA) >
```

*"|": **Choice***
The above box address and the original street address are valid w.r.t. this "|"-DTD

Document Type Tree:



Phone & Fax Example: Address Variant

XML Markup:

```
<address>
  <name>Xaver M. Linde</name>
  <street>Wikingerufer 7</street>
  <town>10555 Berlin</town>
  <phone>030/1234567</phone>
  <phone>030/1234568</phone>
  <fax>030/1234569</fax>
</address>
```

Prolog Term:

```
address(
  name("Xaver M. Linde"),
  street("Wikingerufer 7"),
  town("10555 Berlin"),
  phone("030/1234567"),
  phone("030/1234568"),
  fax("030/1234569")
)
```

Node-Labeled, (Left-to-Right-)Ordered Element Tree:

"+"/"*"-Repetitive-Phone & -Fax Example: Document Type Definition and Tree

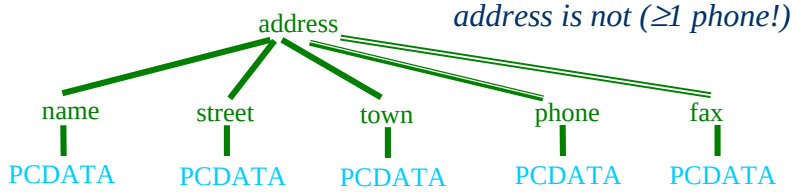
Document Type Definition (DTD):

```
<!ELEMENT address  (name, street, town, phone+, fax*) >
<!ELEMENT name     (#PCDATA) >
<!ELEMENT street   (#PCDATA) >
<!ELEMENT town     (#PCDATA) >
<!ELEMENT phone    (#PCDATA) >
<!ELEMENT fax      (#PCDATA) >
```

"+"/"*": One/Zero or More

The above two-phone/one-fax address is valid w.r.t. this "+"/""-DTD but the original no-phone/no-fax address is not (≥1 phone!)*

Document Type Tree:



Country Example: Address Variant

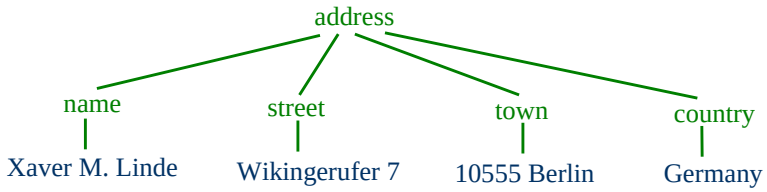
XML Markup:

```
<address>
  <name>Xaver M. Linde</name>
  <street>Wikingerufer 7</street>
  <town>10555 Berlin</town>
  <country>Germany</country>
</address>
```

Prolog Term:

```
address(
  name("Xaver M. Linde"),
  street("Wikingerufer 7"),
  town("10555 Berlin"),
  country("Germany")
)
```

Node-Labeled, (Left-to-Right-)Ordered Element Tree:



"?"-Optional-Country Example: Document Type Definition and Tree

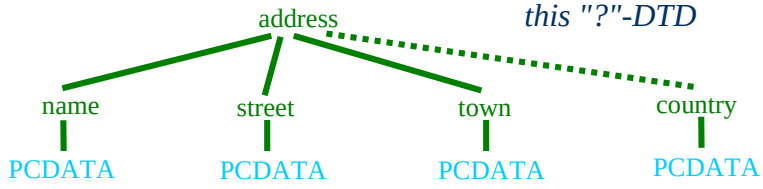
Document Type Definition (DTD):

```
<!ELEMENT address    (name, street, town, country?) >
<!ELEMENT name       (#PCDATA) >
<!ELEMENT street     (#PCDATA) >
<!ELEMENT town       (#PCDATA) >
<!ELEMENT country    (#PCDATA) >
```

"?": One or Zero

The above country address and the original countriless address are valid w.r.t. this "?"-DTD

Document Type Tree:



Country Address: A Complete XML Document Referring to an External DTD

XML Document (just ASCII, e.g. stored in a file):

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE address SYSTEM "country-address.dtd">
<address>
  <name>Xaver M. Linde</name>
  <street>Wikingerufer 7</street>
  <town>10555 Berlin</town>
  <country>Germany</country>
</address>
```

The *XML declaration* uses standalone attribute with "no" value: DTD import

The *DOCument TYPE declaration* names the *root element* address and, after the *SYSTEM keyword*, refers to an *external DTD* "country-address.dtd" (or, at some absolute URL, to an "http://www.test.org/country-address.dtd")

XML

Horn Logic Markup Languages

HornML

Herbrand Terms: Individual Constants, Variables, Flat Ground Structures, ...

Representation of Herbrand terms in XML as <ind> and <struc> elements (<var> similar):

- *Individual constant* for channel-tunnel between Britain and France: element <ind>channel-tunnel</ind>
- *Ground structure*

  undersea-connection(britain,france)

  is <struc> element with embedded element for constructor, followed by elements for argument terms:

```
<struc>
  <constructor>undersea-connection</constructor>
  <ind>britain</ind>
  <ind>france</ind>
</struc>
```


Herbrand Terms: ..., Nested Ground Structures

Embedded <struc> elements:

service-tunnel(
  undersea-connection(
    britain,
    surrounded-country(
      belgium,
      luxembourg,
      germany,
      switzerland,
      italy,
      spain)))

```
<struc>
  <constructor>service-tunnel</constructor>
  <struc>
    <constructor>undersea-connection</constructor>
    <ind>britain</ind>
    <struc>
      <constructor>surrounded-country</constructor>
      <ind>belgium</ind>
      <ind>luxembourg</ind>
      <ind>germany</ind>
      <ind>switzerland</ind>
      <ind>italy</ind>
      <ind>spain</ind>
    </struc>
  </struc>
</struc>
```

Interim Discussion: Tag and Type

- In Prolog not clear, in isolation, that channel-tunnel is individual constant, whereas service-tunnel is constructor
- In XML, as in strongly typed LP languages, made explicit - however at every occurrence of a symbol
- Example gives impression of self-description advantage - but also 'space requirement' - of this generous application of "syntactic sugar"

Horn Clauses: Relation Symbol Applications

Predicate or *relation symbol* in XML is *<relator> element.*
For example, relation symbol travel is
<relator>travel</relator>

Relation symbol application to terms is labeled with *<relationship> element.*
Application travel(john,channel-tunnel)
on two individual constants thus is

```
<relationship>
  <relator>travel</relator>
  <ind>john</ind>
  <ind>channel-tunnel</ind>
</relationship>
```

Horn Clauses: Facts

Hence, Horn fact can be asserted as <hn> element that possesses <relationship> elements as subelements

In the example, the Prolog fact

travel(john,channel-tunnel).

becomes

```
<hn>
  <relationship>
    <relator>travel</relator>
    <ind>john</ind>
    <ind>channel-tunnel</ind>
  </relationship>
</hn>
```

HORNML

Horn Clauses: Rules

Then, Horn rule is asserted as *<hn> Element* that has a head <relationship> element followed by at least one body <relationship> element
So, above example generalized to Prolog rule

travel(Someone,channel-tunnel) :- carry(eurostar,Someone).

and rewritten as

```
<hn>
  <relationship>
    <relator>travel</relator>
    <var>someone</var>
    <ind>channel-tunnel</ind>
  </relationship>
  <relationship>
    <relator>carry</relator>
    <ind>eurostar</ind>
    <var>someone</var>
  </relationship>
</hn>
```

HORNML

Attributes for Extended Logics

Nested elements - trees - allow representation of arbitrary information, but in some situations lead to unnecessarily deeply/widely nested representations

Therefore XML *attributes*:
Start-Tag is 'attributed' with n attribute-value pairs a_i=v_i
Element: <*tag* a_1=v_1 ... a_n=v_n> . . . </*tag*>

Helpful Prolog uses of XML attributes are arity labelings of relation symbols such as our binary relation symbol travel: Prolog's travel/2 in XML with an arity attribute becomes
<relator arity="2">travel</relator>

Analogously, annotations become possible on arbitrary element levels: mode declarations for logic variables, determinism specifications for clauses or procedures, and context conditions for entire knowledge bases

ID and IDREF

Attribute types ID and IDREF for naming and referencing of elements

ID-typed value must uniquely identify an element and IDREF-typed value must occur as ID value of an element

E.g., clause can be named (in a separate knowledge base):

```
<hn id="john-channel">
  <relationship>
    <relator>travel</relator>
    <ind>john</ind>
    <ind>channel-tunnel</ind>
  </relationship>
</hn>
```

ID and *IDREF*

Now a “modal Prolog fact”

belief(mary,travel(john,channel-tunnel)).

can access the "john-channel" assertion:

```
<hn>
  <relationship>
    <relator>belief</relator>
    <ind>mary</ind>
    <prop idref="john-channel"/>
  </relationship>
</hn>
```

Propositional argument of the belief operator written as <prop idref="john-channel"/> (XML abbreviation of empty elements <*tag* ...> </*tag*> to <*tag* .../>)

Also disbelief fact has "john-channel" access with idref: ID/IDREF “break out of the tree” and enable ‘sharing’

HornLogic in XML

DTDs: Elements as Derivation Trees

Up to now: Examples for Horn Logic in XML etc.
Now: General language definition

XML's *Document type definitions (DTDs)* initially only as *ELEMENT declarations* for **non-attributed elements**

For nonterminals: DTD ≈ ordinary context-free grammar in modified (EBNF) notation

For terminals: Usually arbitrary permutations of the base alphabet ("PCDATA") instead of fixed terminal sequences

DTD grammar derives context-free word patterns: derivation trees themselves - linearized through brackets - as generated result

XML element is *valid* with respect to DTD: can be generated from DTD as linearized derivation tree

HornLogic in XML

DTDs: Defining Horn Logic in XML

Syntactic ELEMENT declaration of Horn logic as a knowledge base (kb) of zero or more Horn clauses (hn*):

```
<!ELEMENT kb           (hn*) >
<!ELEMENT hn           (relationship, relationship*) >
<!ELEMENT relationship (relator, (ind | var | struc)*) >
<!ELEMENT struc        (constructor, (ind | var | struc)*) >
<!ELEMENT relator      (#PCDATA) >
<!ELEMENT constructor  (#PCDATA) >
<!ELEMENT ind          (#PCDATA) >
<!ELEMENT var          (#PCDATA) >
```

Note struc recursion!

HOrnML

DTDs: Generation of the Example Rule (1)

(Start-)symbol kb brackets derived clause(s) as linearized start-tag/end-tag-tree representation <kb> . . . </kb>:

kb

```
<kb> hn* </kb>
.  .  .
<kb> <hn> relationship relationship </hn> </kb>
.  .  .
<kb>
 <hn>
  <relationship>
   <relator>#PCDATA</relator> <var>#PCDATA</var> <ind>#PCDATA</ind>
  </relationship>
  <relationship>
   <relator>#PCDATA</relator> <ind>#PCDATA</ind> <var>#PCDATA</var>
  </relationship>
 </hn>
</kb>
```

HOrnML

DTDs: Generation of the Example Rule (2)

```
<hn>
  <relationship>
    <relator>travel</relator>
    <var>someone</var>
    <ind>channel-tunnel</ind>
  </relationship>
  <relationship>
    <relator>carry</relator>
    <ind>eurostar</ind>
    <var>someone</var>
  </relationship>
</hn>
```

HOrnML

Attribute DTDs (1)

DTDs for **attributed elements**: *ATTLIST declarations*, which associate element name with an attribute name plus attribute type and possible occurrence indication

1st Example: declare the relator attribute arity as CDATA-typed (cf. #PCDATA) and occurring optionally (#IMPLIED):

```
<!ATTLIST relator arity  CDATA  #IMPLIED >
```

2nd Example (Preparation): define the extended Horn logic with (named hn clauses and) embedded propositions:

```
<!ELEMENT relationship (relator, (ind|var|struc|prop)*) >

<!ELEMENT prop EMPTY >
```

HOrnML

Attribute DTDs (2)

2nd Example (Execution): Append an ATTLIST declaration that specifies, for the hn respectively prop elements, the attributes
id - as optional ID type - respectively
idref - as mandatory IDREF type:

```
<!ATTLIST hn      id     ID      #IMPLIED >

<!ATTLIST prop    idref  IDREF   #REQUIRED >
```

With entire DTD now, e.g., earlier "john-channel"-named fact and its accessing facts can be generated

Horn Queries in XML Notation

Assume fact:

```
<hn>
  <relationship>
    <relator>carry</relator>
    <ind>eurostar</ind>          carry(eurostar,fred).
    <ind>fred</ind>
  </relationship>
</hn>
```

A Horn-logic interpreter can use it to answer this query:

```
<relationship>
  <relator>carry</relator>
  <ind>eurostar</ind>            carry(eurostar,Someone)
  <var>someone</var>
</relationship>
```

by binding

```
<var>someone</var>             Someone
 to
<ind>fred</ind>                fred
```

Horn Queries in XML-QL Implementation

Assume the fact is extended by prem(ises)/arity/pos(ition) attributes:

```
<hn prem='0'>
  <relationship arity='2'>
    <relator>carry</relator>
    <ind pos='1'>eurostar</ind>
    <ind pos='2'>fred</ind>
  </relationship>
</hn>
```

Then in XML-QL the query can be implemented like this:

```
WHERE
  <hn prem='0'>
    <relationship arity='2'>
      <relator>carry</relator>
      <ind pos='1'>eurostar</ind>
      <ind pos='2'>$someone</ind>
    </relationship>
  </hn>
CONSTRUCT
  <binding> <someone>$someone</someone> </binding>
```


Horn Inferences in XML Notation (1)

With carry basis fact

carry(eurostar,fred).

a rule is usable to dynamically derive travel assertions as needed, without having to store them all-inclusively and statically:

travel(Someone,channel-tunnel) :- carry(eurostar,Someone).

That is, its earlier XML version is useable by a Horn-logic interpreter for inferential queries like

travel(fred,Where) ⇒ (Someone=fred, Where=channel-tunnel) carry(eurostar,Someone) ⇒ (Where=channel-tunnel) true

Horn Inferences in XML Notation (2)

Rule:

```
<hn>
  <relationship>
    <relator>travel</relator>
    <var>someone</var>
    <ind>channel-tunnel</ind>
  </relationship>
  <relationship>
    <relator>carry</relator>
    <ind>eurostar</ind>
    <var>someone</var>
  </relationship>
</hn>
```

Fact:

```
<hn>
  <relationship>
    <relator>carry</relator>
    <ind>eurostar</ind>
    <ind>fred</ind>
  </relationship>
</hn>
```

3-Step Animation:

```
<relationship>
  <relator>travel</relator>
  <ind>fred</ind>
  <var>where</var>
</relationship>
```

HornML

Horn Inferences in XML Notation (2)

Rule:

```
<hn>
  <relationship>
    <relator>travel</relator>
    <var>someone</var>
    <ind>channel-tunnel</ind>
  </relationship>
  <relationship>
    <relator>carry</relator>
    <ind>eurostar</ind>
    <var>someone</var>
  </relationship>
</hn>
```

Fact:

```
<hn>
  <relationship>
    <relator>carry</relator>
    <ind>eurostar</ind>
    <ind>fred</ind>
  </relationship>
</hn>
```

3-Step Animation:

```
<relationship someone="fred" where="channel-tunnel">
  <relator>carry</relator>
  <ind>eurostar</ind>
  <var>someone</var>
</relationship>
```

HornML

Horn Inferences in XML Notation (2)

Rule:

```
<hn>
  <relationship>
    <relator>travel</relator>
    <var>someone</var>
    <ind>channel-tunnel</ind>
  </relationship>
  <relationship>
    <relator>carry</relator>
    <ind>eurostar</ind>
    <var>someone</var>
  </relationship>
</hn>
```

Fact:

```
<hn>
  <relationship>
    <relator>carry</relator>
    <ind>eurostar</ind>
    <ind>fred</ind>
  </relationship>
</hn>
```

3-Step Animation:

```
<con where="channel-tunnel">
  true
</con>
```

HornML

Horn Inferences: SLD-Resolution, XML-QL Implementation, Open World

- This inference is carried out as an SLD-resolution step
- The procedural semantics of SLD-resolution can be used
- An XML-QL implementation seems possible as for queries

If distribution of the clauses over different documents in the Web is assumed, in this “open world” logical completeness, in particular, can hardly still be asked for

HornML

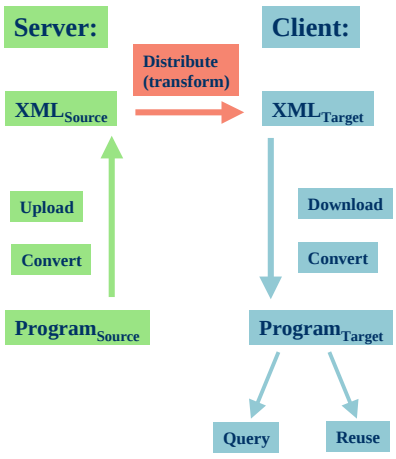
Relational-Functional Markup Language

RFML

A 10-Step Strategy to Publish and Reuse Declarative Programs as XML Markups

RFML

1. **Specify the declarative programming language** through an XML document type definition (DTD)
2. **Convert** any to-be-published declarative program **from its source syntax to an XML document** according to the DTD
3. **Upload** such an XML document to a Web server for publication
4. Also offer the declarative programs for **server-side querying (e.g. CGI)** and advertise their XML-document version to search engines etc., ideally using **metadata markup (e.g. RDF/XML)**
5. **Distribute** these documents to requesting clients **via standard Web protocols (e.g. HTTP)**
6. **If necessary, transform** such an XML document **to** a declarative target language with a **different DTD**, possibly using an (XSLT) stylesheet
7. **Download** any requested XML document at the client site
8. **Convert** this XML document **to** the client's **target** syntax, possibly using a (CSS or XSLT) stylesheet
9. **Query** the **target version** via the client's program interpreter and optionally download the server's source-program interpreter (once) for client-side querying, ultimately as a browser plug-in
10. **Reuse** the target version, say in existing programs



Note: **Program**$_{Source}$ may be identical to **Program**$_{Target}$

Cross-Fertilizations of XML and Declarative Programming Languages

- Separate vs. joint assertion and query languages:
  - XML: Still separate schema and query of elements
  - DPL: Mostly joint storage and retrieval of clauses
- Generating XML markup from more compact special-purpose notations (and vice versa)
- XML validators and DPL compilers
- XML stylesheets and DPL transformers
- Specification, correctness, and efficiency technology
- Early case study with the declarative language RFML (Relational-Functional Markup Language)

RFML

Basics of the Relational-Functional Markup Language RFML

- Much of Web knowledge constitutes definitions of relations and functions
- Kernel of Relational-Functional language suited for XML knowledge markup:
  - Uniform, rather small language
  - Sufficient expressive power for practical use
- RFML is an XML application for integrated relational-functional information
- Relational (hn) and functional (ft) clauses together define a unified notion of operators
- RFML DTD small and open to various extensions

RFML

Relational Facts: From Tables to Prolog

Collect data on consumer behavior in ...

Relational Table:

satisfied(	Customer,	Item,	Price	)
	john	wine	17.95	
	peter	beer	06.40	

Prolog (Ground) Facts:

satisfied(	Customer,	Item,	Price	)
satisfied(	john,	wine,	17.95	).
satisfied(	peter,	beer,	06.40	).

RFML

Relational Facts: From Prolog to RFML

Prolog (Ground) Facts:

satisfied(john,wine,17.95). satisfied(peter,beer,6.40).

RFML (Ground) Markup:

```
<hn>
  <pattop>
    <con>satisfied</con>
    <con>john</con>
    <con>wine</con>
    <con>17.95</con>
  </pattop>
</hn>
```

```
<hn>
  <pattop>
    <con>satisfied</con>
    <con>peter</con>
    <con>beer</con>
    <con>6.40</con>
  </pattop>
</hn>
```

RFML

Relational Rules: From Prolog to RFML

Infer data on consumer behavior via ...

Prolog (Non-Ground) Rule:

satisfied(C,I,P) :- buy(week1,C,I,P), buy(week2,C,I,P).

RFML (Non-Ground) Markup:

```
<hn>
 <pattop>
  <con>satisfied</con><var>C</var><var>I</var><var>P</var>
 </pattop>
 <callop>
  <con>buy</con><con>week1</con><var>C</var><var>I</var><var>P</var>
 </callop>
 <callop>
  <con>buy</con><con>week2</con><var>C</var><var>I</var><var>P</var>
 </callop>
</hn>
```


Functional Facts: From Unconditional Equations to RFML

Discriminate on payment method via ...

Unconditional (Ground) Equations:

pay(john,fred,17.95) = cheque pay(peter,fred,6.40) = cash

RFML (Ground) Markup:

```
<ft>
  <pattop>
    <con>pay</con>
    <con>john</con>
    <con>fred</con>
    <con>17.95</con>
  </pattop>
  <con>cheque</con>
</ft>
```

```
<ft>
  <pattop>
    <con>pay</con>
    <con>peter</con>
    <con>fred</con>
    <con>6.40</con>
  </pattop>
  <con>cash</con>
</ft>
```

Functional Queries: Joint Assertion and Query Language

The pay function can be queried (non-ground) **directly** via a callop markup:

```
<callop>
  <con>pay</con>
  <con>john</con>
  <var>merchant</var>
  <var>price</var>
</callop>
```

binding the two variables to the corresponding constants in the definition pattern and returning the constant 'cheque'

Same **indirectly** as the right side of a conditional equation ...

RFML

Functional Rules: From Conditional Equations to Relfun

Predict consumers' acquisition behavior via ...

Conditional (Non-Ground) Equation:

acquire(Customer,Merchant,Item,Price) =
pay(Customer,Merchant,Price)
if satisfied(Customer,Item,Price)

Relfun (Non-Ground) Footed Rule:

```
acquire(Customer,Merchant,Item,Price) :-
  satisfied(Customer,Item,Price) &
  pay(Customer,Merchant,Price).
```

RFML

Functional Rules: From Relfun to RFML

Relfun (Non-Ground) Footed Rule:

acquire(C,M,I,P) :- satisfied(C,I,P) & pay(C,M,P).

RFML (Non-Ground) Markup:

```
<ft>
 <pattop>
  <con>acquire</con><var>c</var><var>m</var><var>i</var><var>p</var>
 </pattop>
 <callop>
  <con>satisfied</con><var>c</var><var>i</var><var>p</var>
 </callop>
 <callop>
  <con>pay</con><var>customer</var><var>m</var><var>p</var>
 </callop>
</ft>
```

RFML

Relational-Functional Computations: "What Items John Buys, and How"

A query of the acquire function now leads to the following RFML computation (4-step animation):

```
<callop>
  <con>acquire</con>
  <con>john</con>
  <con>fred</con>
  <var>item</var>
  <con>17.95</con>
</callop>
```

It binds the variable 'item' to the constant 'wine' (RFML bindings represented as XML attributes) and returns the constant 'cheque'

RFML

Relational-Functional Computations: "What Items John Buys, and How"

A query of the acquire function now leads to the following RFML computation (4-step animation):

```
<callop>
  <con>satisfied</con>
  <con>john</con>
  <var>item</var>
  <con>17.95</con>
</callop>
```

&

```
<callop>
  <con>pay</con>
  <con>john</con>
  <con>fred</con>
  <con>17.95</con>
</callop>
```

It binds the variable 'item' to the constant 'wine' (RFML bindings represented as XML attributes) and returns the constant 'cheque'

RFML

Relational-Functional Computations: "What Items John Buys, and How"

A query of the acquire function now leads to the following RFML computation (4-step animation):

```
<con item=wine>
  true
</con>
```

&

```
<callop>
  <con>pay</con>
  <con>john</con>
  <con>fred</con>
  <con>17.95</con>
</callop>
```

It binds the variable 'item' to the constant 'wine' (RFML bindings represented as XML attributes) and returns the constant 'cheque'

RFML

Relational-Functional Computations: “What Items John Buys, and How”

A query of the acquire function now leads to the following RFML computation (4-step animation):

```
<con item=wine>
  cheque
</con>
```

It binds the variable ’item’ to the constant ’wine’ (RFML bindings represented as XML attributes) and returns the constant ’cheque’

The RFML DTD (1)

```
<!-- ENTITIES use non-terminals of Relfun grammar (Boley 1999) ’untagged’,  -->
<!-- e.g. term ::= con | var | anon | struc | tup, just specifying, say,    -->
<!-- <var> X </var> term instead of nesting <term> <var> X </var> </term>   -->

<!ENTITY % variable      "(var | anon)" >
<!ENTITY % appellative   "(con | %variable; | struc)" >
<!ENTITY % term          "(%appellative; | tup)" >

<!-- ELEMENTS use non-terminals of Relfun grammar ’tagged’, so var ::= ...  -->
<!-- itself becomes <var> X </var>                                           -->

<!-- rfml is the document root, the possibly empty knowledge-base top-level -->
<!-- of hn or ft clauses:                                                    -->

<!ELEMENT rfml          (hn | ft)* >

<!-- hn clauses are a pattop before zero (facts) or more terms or callop’s; -->
<!-- ft clauses are a pattop before at least one term or callop (the foot): -->

<!ELEMENT hn            (pattop, (%term; | callop)*) >
<!ELEMENT ft            (pattop, (%term; | callop)+) >

<!-- a pattop clause head is an operator appellative and a (rest) pattern:  -->

<!ELEMENT pattop        (%appellative;,
                         (%term;)*,
                         (rest, (%variable; | tup))?) >
```

The RFML DTD (2)

```
<!-- a callop clause body premise or foot is a (nested) operator call:      -->

<!ELEMENT callop        ((%appellative; | callop),
                         (%term; | callop)*,
                         (rest, (%variable; | tup | callop))?) >

<!-- a struc is a constructor appellative with argument terms (and a rest): -->

<!ELEMENT struc         (%appellative;,
                         (%term;)*,
                         (rest, (%variable; | tup))?) >

<!-- a tup is a list of terms (zero or more), perhaps followed by a rest:   -->

<!ELEMENT tup           ((%term;)*,
                         (rest, (%variable; | tup))?) >

<!-- con and var are just parsed character data (character permutations):   -->

<!ELEMENT con           (#PCDATA)>
<!ELEMENT var           (#PCDATA)>

<!-- anon (Relfun: "_") and rest (Relfun: "|") are always-empty elements:   -->

<!ELEMENT anon          EMPTY >
<!ELEMENT rest          EMPTY >
```


RFML Summary

- RFML combines relational-functional knowledge-representation and declarative-programming languages on the Web
- It has been implemented as a (Web-)output syntax for declarative knowledge bases and computations
- RFML stylesheets for Prolog, Relfun, and other languages are under development
- Further descriptions, examples, the DTD, and download information are available at http://www.relfun.org/rfml/

Simple HTML/XML Ontology Extensions

SHOE

SHOE Basics

SHOE (Simple HTML Ontology Extensions)
(http://www.cs.umd.edu/projects/plus/SHOE/)

provides distributed **ontologies** consisting of

Categories: Organized hierarchically, with multiple inheritance, for classifying instances

Relationship rules: Horn clauses (the first well-known Horn language publishing KBs in the Web)

SHOE originally specified in HTML (before the definition of XML); meanwhile also specified as an XML DTD

SHOE

Instances (Individuals) as URLs/URIs

SHOE

Individual constants in SHOE are represented through an (official) URL/URI

In the earlier Horn-rule example there appear two individuals, which could be represented in SHOE, e.g., as

eurostar	=	http://www.eurostar.com/
channel-tunnel	=	http://www.eurotunnel.com/

A SHOE Rule

SHOE

With these URLs/URIs, the rule in SHOE becomes

```
<DEF-INFERENCE
   DESCRIPTION="travel(?someone,http://www.eurotunnel.com/) if
                carry(http://www.eurostar.com/,?someone)">
   <INF-IF>
      <RELATION NAME="carry">
         <ARG POS="1" VALUE="http://www.eurostar.com/">
         <ARG POS="2" VALUE="someone" USAGE="VAR">
      </RELATION>
   </INF-IF>
   <INF-THEN>
      <RELATION NAME="travel">
         <ARG POS="1" VALUE="someone" USAGE="VAR">
         <ARG POS="2" VALUE="http://www.eurotunnel.com/">
      </RELATION>
   </INF-THEN>
</DEF-INFERENCE>
```

XML-Based Ontology Exchange Language

XOL

XOL: XML-based Ontology Exchange Language
(by Peter Karp, Vinay Chaudhri, Jerome Thomere, SRI)

XOL

- A language for specifying ontologies
- A language for exchanging ontologies
- Can also be used for exchange of databases
- Frame-based semantic model: OKBC-Lite
- Expressive power similar to Ontolingua
- XML-based syntax (kernel DTD)

Essential Form of an XOL File

```
<module>
   . . .
   <class>.....</class>
   <class>.....</class>

   <slot>.....</slot>
   <slot>.....</slot>

   <individual>.... </individual>
   <individual>.... </individual>
</module>
```


Module-Header Definition

```
<module>
   <name>Genes</name>
   <version>1.2</version>
   <documentation>An Ontology for the functional
   classification of genes</documentation>
   . . .
</module>
```

Class Definition

```
<class>
   <name>Genes</name>
   <documentation>The class of all Genes.  Each
   subclass of class Genes describes a specific category
   of gene function.</documentation>
</class>

<class>
   <name>GC001</name>
   <subclass-of>Genes</subclass-of>
</class>
```

XOL

Slot Definition

```
<slot>
   <name>common-name</name>
   <documentation>A string that encodes the official
   name of the entity.</documentation>
   <domain>Genes</domain>
   <slot-cardinality>1</slot-cardinality>
   <slot-value-type>string</slot-value-type>
</slot>
```

XOL

Individual Definition

```
<individual>
  <name>EG10115</name>
  <documentation>The trpC gene</documentation>
  <instance-of>GC001</instance-of>
  <slot-values>
    <name>common-name</name>
    <value>trpC</value>
  </slot-values>
  ....
</individual>
```

XOL

XML Namespaces

XML Namespaces and Programming-Language Modules

- XML namespaces are akin to namespaces, packages, and modules in programming languages
- Disambiguation of tag–and attribute–names from different XML applications (“spaces”) through different prefixes
- A *prefix* is separated from the local *name* by a “:”, obtaining *prefix*:*name* tags
- Namespaces constitute a layer on top of XML 1.0, since *prefix*:*name* is again a valid tag name and namespace bindings are ignored by some tools

Namespace Bindings

- Prefixes are bound to namespace URIs by attaching an xmlns:*prefix* attribute to the prefixed element or one of its ancestors, *prefix*:$name_1$,..., *prefix*:$name_n$
- The value of the xmlns:*prefix* attribute is a URI, which may or (unlike for DTDs!) may not point to a description of the namespace’s syntax
- An element can use bindings for multiple name-spaces via attributes xmlns:$prefix_1$,..., xmlns:$prefix_m$

Namespaceless Example: Address Variant

Namespaceless XML Markup:

```
<address>
  <name>Xaver M. Linde</name>
  <street>Wikingerufer 7</street>
  <town>10555 Berlin</town>
  <bill>12.50</bill>
  <phone>030/1234567</phone>
  <phone>030/1234568</phone>
  <fax>030/1234569</fax>
  <bill>76.20</bill>
</ address>
```

bill is ambiguous tag (name clash from two XML applications)

Two-Namespace Example: Snail-Mail and Telecoms Address Parts

Namespace XML Markup:

```
<mail:address xmlns:mail="http://www.deutschepost.de/"
              xmlns:tele="http://www.telekom.de/">
  <mail:name>Xaver M. Linde</mail:name>
  <mail:street>Wikingerufer 7</mail:street>
  <mail:town>10555 Berlin</mail:town>
  <mail:bill>12.50</mail:bill>
  <tele:phone>030/1234567</tele:phone>
  <tele:phone>030/1234568</tele:phone>
  <tele:fax>030/1234569</tele:fax>
  <tele:bill>76.20</tele:bill>
</ mail:address>
```

bill disambiguation through mail and tele prefixes

- The root element, mail:address, as well as the children mail:name, mail:street, mail:town, and mail:bill, use the mail prefix, bound to a deutschepost URI
- The tele:phone, tele:fax, and tele:bill children use the tele prefix, bound to a telekom URI

Acquiring and Processing Knowledge Markups

A&P

Acquiring and Processing Knowledge Markups

- acquisition
  - Protégé with XML Plugin
  - Web Onto
- transformation techniques and stylesheet languages
  - CSS
  - XSLT
- query languages
  - XQL and XPath
  - XML-QL

A&P

Acquiring XML Knowledge Bases

Protégé-2000 as an XML Editor

- Represents the latest in a series of interactive tools for knowledge-system development
- Facilitates construction of knowledge bases in a principled fashion from reusable components
- Allows a variety of “plug-ins” to facilitate customization in various dimensions
- One plug-in used for XML import/export, i.e. Protégé proposed as an **XML editor** (also as an RDF[S] editor)

Knowledge-Base Development with Protégé-2000

1. Build or import a domain ontology (a conceptual model of the application area)
2. Custom-tailor GUI for acquisition of content knowledge
3. Elicit content knowledge from application specialists
4. Map domain ontology to appropriate problem solvers for automation of particular tasks
5. Export ontology and content knowledge to target format (OKBC, XML, RDF[S], ...)

Protégé

Protégé as an OKBC-Compliant System (Open Knowledge Base Connectivity)

OKBC:

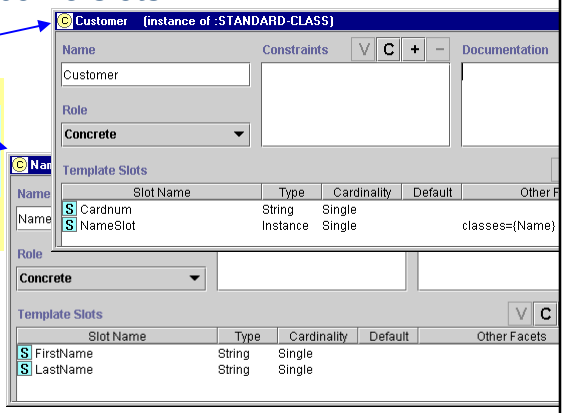
- Standard mechanism for knowledge bases stored as “frames” of classes, slots, facets, instances, ...
- Adopted by several well-known knowledge-representation systems (Ontolingua, LOOM, **Protégé-2000**, XOL)
- Allows Protégé-2000 to be used as an ontology- and knowledge-editing system for any OKBC-compliant server

Protégé

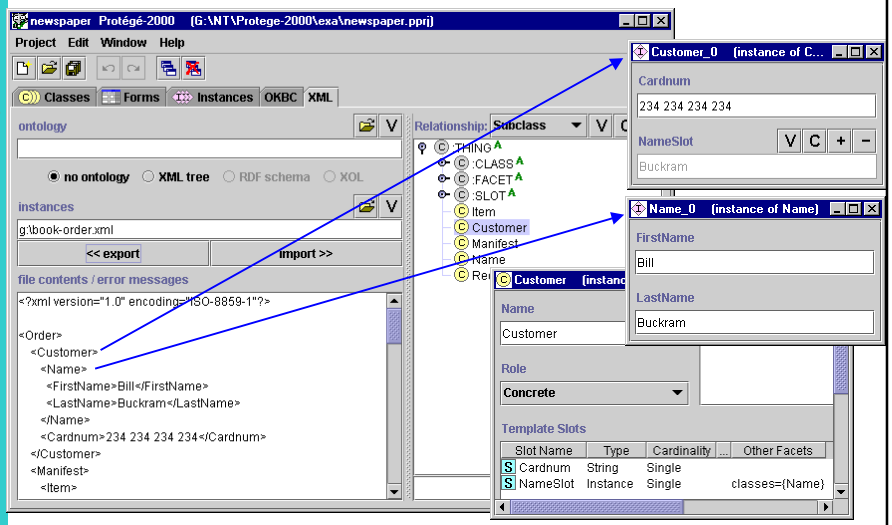
XML Import Strategy

- tag/element names become class names, except "leaves" which become slots
- example:

```
<Customer>
    <Name>
      <FirstName>Bill</FirstName>
      <LastName>Buckram</LastName>
    </Name>
    <Cardnum>234 ...</Cardnum>
</Customer>
```



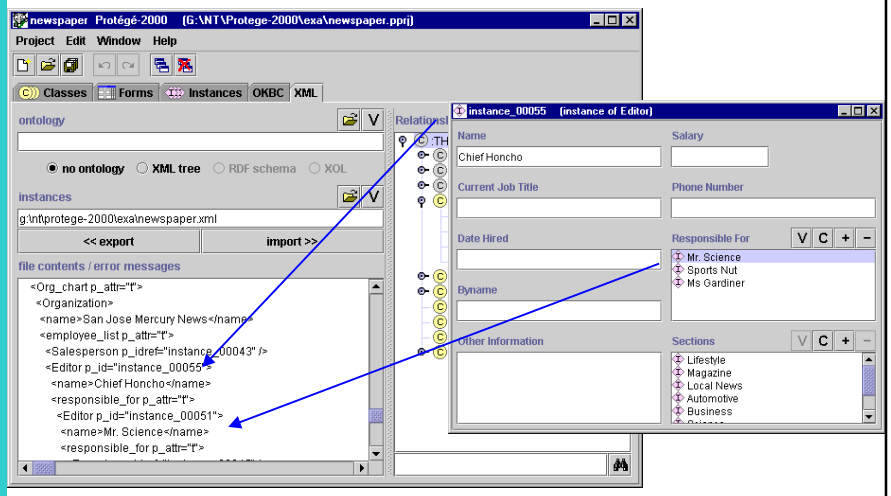
Example (Import): Book Order



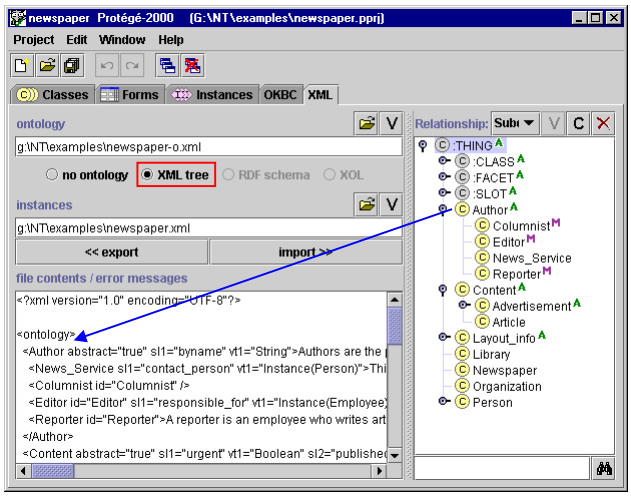
XML Export Strategy

- instances:
  - unreferenced instances become top level elements (cyclic references are handled)
  - classes and slots become tag names
  - objects that are referenced more than once are shared/reused with id/idref
- ontology:
  - as simple XML tree, RDF schema, XOL, … (future work)

Example (Export): Newspaper Instances



Example: Newspaper Ontology As XML Tree



Processing XML

Cascading Style Sheets

- CSS is a language for applying styles such as *bold* and *Arial* to XML elements
- also works with HTML (supported in most modern browsers)
- example (style sheet for poems):

```
poem    { display: block }
title   { display: block; font-size: 16pt; font-weight: bold}
poet    { display: block; margin-bottom: 10px }
stanza  { display: block; margin-bottom: 10px }
verse   { display: block }
```

- attached to XML documents with processing instruction:
  <?xml-stylesheet type="text/css" href="poem.css"?>

XSLT (XSL Transformations)

- XSL (Extensible Stylesheet Language) =
  XSLT + Formatting Objects
- XSLT is a rule-based transformation language for XML documents/trees
- result of a transformation usually is again an XML document, but may also be HTML or plain text
- transformation takes place
  - offline (useful for XML-to-HTML transformation)
  - on server (e.g., with Apache´s Cocoon)
  - in client (browser, preliminary support in Netscape 6 and IE 5.x)

XSLT Example – Input

```
<addresses>

  <address>
    <name>Xaver M. Linde</name>
    <street>Wikingerufer 7</street>
    <town>10555 Berlin</town>
  </address>

  <address>
    <name>John Doe</name>
    <street>42 Gary Cooper Street</street>
    <town>Stanwyck City</town>
  </address>

</addresses>
```


XSLT Example – Stylesheet

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

<xsl:template match="/">
  <html>
    <head><title>Addresses</title></head>
    <body bgcolor="white">
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>

<xsl:template match="address">
  <p>
    <i><xsl:value-of select="name"/></i><br/>
    <xsl:value-of select="street"/><br/>
    <b><xsl:value-of select="town"/></b>
  </p>
</xsl:template>

</xsl:stylesheet>
```

template for document root

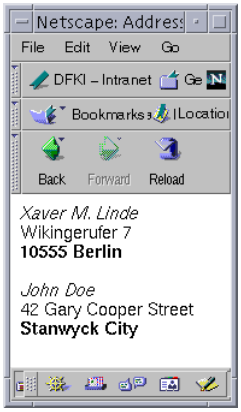
template for address elements

XSLT Example – Output

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
 "http://www.w3.org/TR/REC-html40/strict.dtd">

<html>
 <head><title>Addresses</title></head>
 <body bgcolor="white">
  <p><i>Xaver M. Linde</i><br>
      Wikingerufer 7<br>
      <b>10555 Berlin</b>
  </p>
  <p><i>John Doe</i><br>
      42 Gary Cooper Street<br>
      <b>Stanwyck City</b>
  </p>
 </body>
</html>
```

*(The HTML code was produced with Apache´s Cocoon in HTML mode, hence
 became
)*



XQL and XPath

- XSLT uses XPath to select parts of the input XML documents, e.g.:
  - <xsl:template match="/"> *document root*
  - <xsl:value-of select="name"/> *element „name“*
- syntax is not XML-based, but intended to be used in URLs (XPointer) and XSLT as above
- XQL is a variant of XPath especially designed for querying XML documents
- XPath/XQL expressions *select parts* of XML documents (no transformations!)

XQL Expressions 1

- XQL expressions (relative to a context node):
  - address
    select element nodes with tag name „address“
  - address/name
    select „name“ element nodes *directly* below „address“ element nodes
  - address/name=´John Doe´
    as before, plus the content of the name element must be ´John Doe´
  - book//name
    select „name“ element nodes *anywhere* below „book“ element nodes

XQL Expressions 2

- absolute XQL expressions:
  - /
    document root (the node that contains the document element)
  - /addresses/address/name
  - //name
    „name“ element nodes anywhere in the document
- expressions with attributes (prefixed with @):
  - address/@type
    attribute nodes with name „type“ below „address“ element nodes
  - address/@type=´email´
    as above, plus the value of the „type“ attribute is ´email´

XQL Expressions 3

- filter expressions:
  - address[name]
    select „address“ element nodes with a direct „name“ sub-element node
  - address[name=´John Doe´]
    as above, plus the content of the „name“ element is ´John Doe´
  - address[@type=´email´]
    select „address“ element nodes that have a „type“ attribute whose value is ´email´

XQL

XML-QL

- borrows features of query languages developed by the database research community for semistructured data
- similar to SQL
- (simplified) base construct is
  WHERE *pattern*$_1$ IN *URI*$_1$, ...
  CONSTRUCT *pattern*$_n$
  where the patterns are XML fragments with variables (prefixed with $)
- XML-QL introduces an abbreviated XML syntax for simplicity: <tag>...</> instead of <tag>...</tag>

XML-QL

XML-QL Example 1

- return short addresses (town-city renaming, without street):

```
WHERE
  <address>
    <name>$n</>
    <town>$t</>
  </> IN "www.test.org/addresses.xml"
CONSTRUCT
  <shortaddress>
    <name>$n</>
    <city>$t</>
  </>
```

<street> elements are also matched

```
<address>
   <name>Xaver M. Linde</name>
   <street>Wikingerufer 7</street>
   <town>10555 Berlin</town>
</address>
```

```
<shortaddress>
   <name>Xaver M. Linde</name>
   <city>10555 Berlin</city>
</shortaddress>
```

XML-QL

XML-QL Example 2

- XML-QL allows joining by value:

```
WHERE
  <address>
    <name>$n</>
  </> ELEMENT_AS $a IN "www.test.org/addresses.xml",
  <book>
    <author>$n</>
    <title>$t</>
  </> IN "www.test.org/books.xml"
CONSTRUCT
  <book>
    <title>$t</>
    $a
  </>
```

XML-QL

XML-Based Agent Techniques

XML-based Agent Techniques

- XML-RPC
  - remote procedure calls using HTTP as the transport and XML as the encoding
- DAML
  - DARPA Agent Markup Language
- **FRODO framework**
  - under development at DFKI Kaiserslautern in the Knowledge Management Group
- **Ontobroker**
  - AIFB/University of Karlsruhe

The FRODO Agent Framework

FRODO

- integration of
  - separately developed OM (organizational memory) solutions and legacy (DB) systems
  - DAU (document analysis and understanding) components (e.g., for wrapping)
- distributed problem solving
- based on
  - declarative knowledge representation
  - agents / speech acts / protocols
  - Internet-enabled: HTTP, XML, RDF, ...

Involved XML Technologies

FRODO

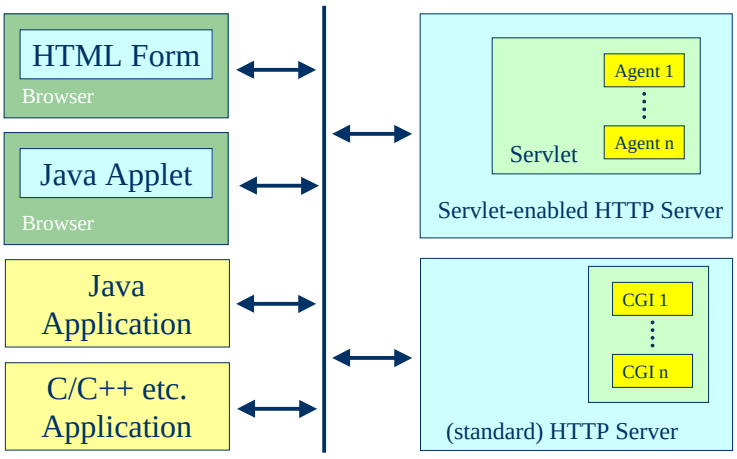
- FIPA-like agents communicate with XML messages
- communication via HTTP
- involved XML technologies:
  - XSLT for message transformation and information extraction
  - RDF for representing meta-information
  - RDF Schema for representing ontologies
  - XML/RDF-based query and transformation languages for distributed inferences
- first prototypical realization of an RDF/RDF Schema query agent (based on an F-Logic implementation for RDF: SiLRI, Karlsruhe / Stanford)

Agents Communicate Via XML Messages

```
<message type="message-type"
         sender="sender-url"
         receiver="receiver-url"
         additional-information >
  contents
</message>
```

- *message types*: inform, request, agree, cancel, confirm, disconfirm, subscribe, ... (speech acts, FIPA)
- *additional information*: reply-with, in-reply-to, language, ontology, reply-by, protocol, ...
- *contents*: XML “forest”

Message Exchange is Based on Internet Techniques

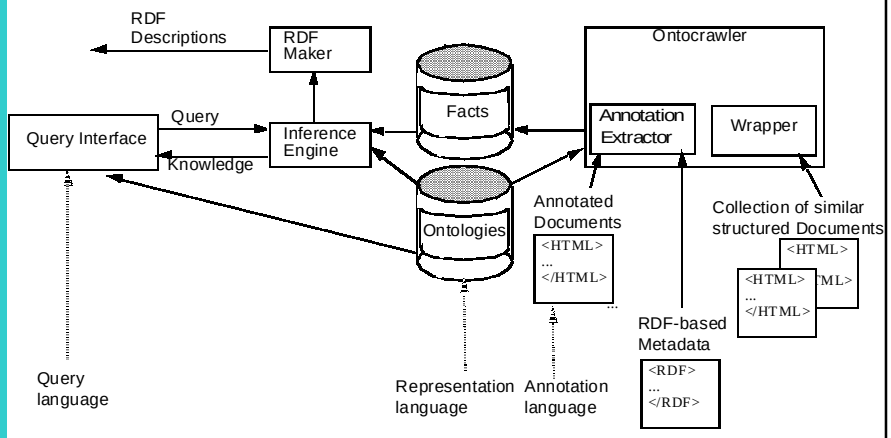


Simple Integration of Foreign Software Components is Enabled

- all foreign software components which are designed as CGI programs are immediately usable as agents (no explicit wrappers needed)
- access to agents simple for all software components that have access to a HTTP library (Java, C, C++, ...)

Ontobroker Application

Ontobroker/On2broker (AIFB/University of Karlsruhe)

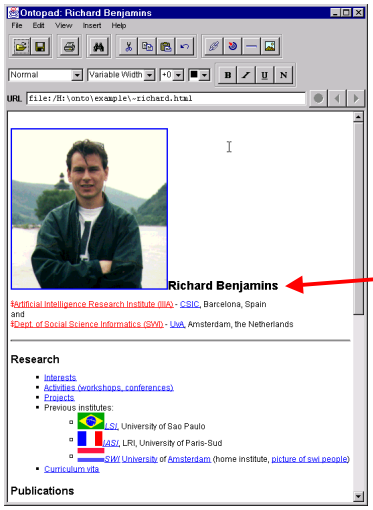


Example Ontology

Concept Hierarchy	Slot Definitions	Rules

Object[].
Person :: Object.
Employee :: Person.
Researcher :: Employe.
Publication::Object.
	Person[
	firstName =>> STRING;
	lastName =>> STRING;
	eMail =>> STRING;
	...
	publication =>> Publication].
	Employee[
	affiliation =>> Organization;
	...].
	Researcher[
	researchInterest=>>ResearchTopic;
	cooperatesWith=>> Researcher].
	Publication[
	author =>> Person;
	title =>> STRING;
	year =>> NUMBER;
	abstract =>> STRING].
		FORALL Person1, Person2
		Person1:Researcher
		[cooperatesWith ->> Person2]
		<-
		Person2:Researcher
		[cooperatesWith ->> Person1].
		
		FORALL Person1, Publ1
		Publ1:Publication
		[author ->> Person1]
		<->>
		Person1:Person
		[publication ->> Publ1].
		
		FORALL O,C,A,V,T
		V:T
		<-
		C[A=>>T] AND O:C[A->>V].

Annotated HTML Pages



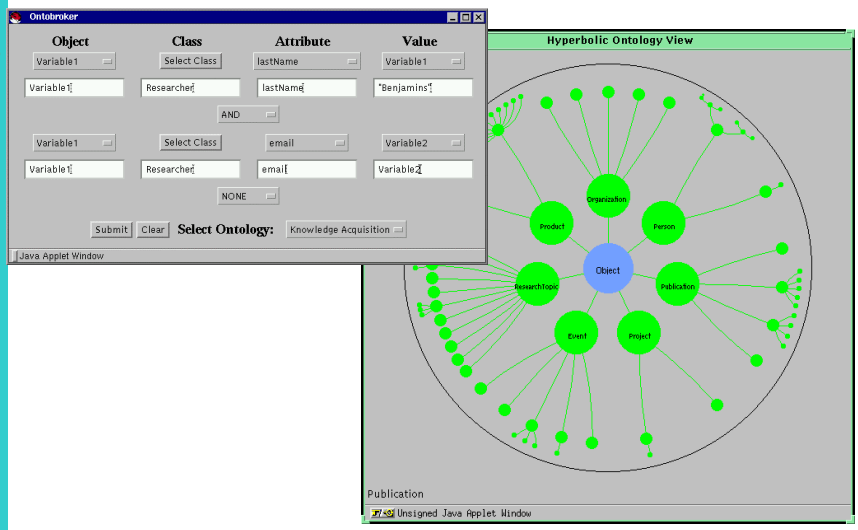

```
<html>
<head><TITLE> Richard Benjamins </TITLE>
<a onto="page:Researcher"> </a>
</head>

<H1> <A HREF="pictures/id-rich.gif">
<IMG  align=middle SRC="pictures/richard.gif"></A>
<a onto="page[photo=href]"
HREF="http://www.iiia.csic.es/~richard/pictures/richard.gif" ></a>

<a onto="page[firstName=body]">Richard</a>
<a onto="page[lastName=body]">Benjamins </a>
</h1> <p>

<A onto="page[affiliation=body]" HREF="#card">
Artificial Intelligence Research Institute (IIIA)</A> -
<a href="http://www.csic.es/">CSIC</a>, Barcelona, Spain <br>
and <br>
<A onto="page[affiliation=body]" HREF="http://www.swi.psy.uva.nl/">
Dept. of Social Science Informatics (SWI)</A>
-
<A HREF="http://www.uva.nl/uva/english/">UvA</A>,
Amsterdam, the Netherlands
```


Ontobroker Query Tool



Resource Description Framework

RDF

Disclaimer

"There is nothing interesting in what we are doing. The only interesting thing is the scale upon which we are attempting to do it."

R.V. Guha, Epinions

RDF

Outline

- Motivation: Why XML is not enough
- Introduction to RDF
  - Requirements for KR on the Web
  - The RDF Data Model
  - RDF Schema
- Extensions of RDF(S)
- Tools for RDF and RDF Schema
  - Parser, Query, and Inference Engines

Why The Shift Towards More Semantics?

- Information Overload
  - Information on the Web currently aiming at Human Consumption
  - Information Consumption is to time consuming
- Search Engines fail more and more
  - combined coverage is less than 42% of the HTML-Web
- Data Interchange growing (e.g. B2B)
  - needs a common semantics

XML?

Extensible Markup Language (XML) Revisited

- Key idea: separate structure from presentation
- XML DTDs or Schema define document structure
- Replace HTML with two things
  - A domain specific markup language (defined in XML)
  - A map from that markup language to HTML (defined using XSL)
- DTD enables document recipients to tell whether they've received a well-formed document
  - Gives a minimal level of validation

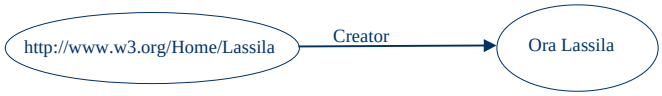
Why XML is Not Enough

- Main advantage of using XML is reusing the parser and document validation
- Many different possibilities to encode a domain of discourse
- Leads to difficulties when understanding of foreign documents is required

==> Next step: separate content from structure!

Encoding of Knowledge: Example

"The Creator of the Resource "http://www.w3.org/Home/Lassila" is Ora Lassila



Endless encoding possibilities in XML:

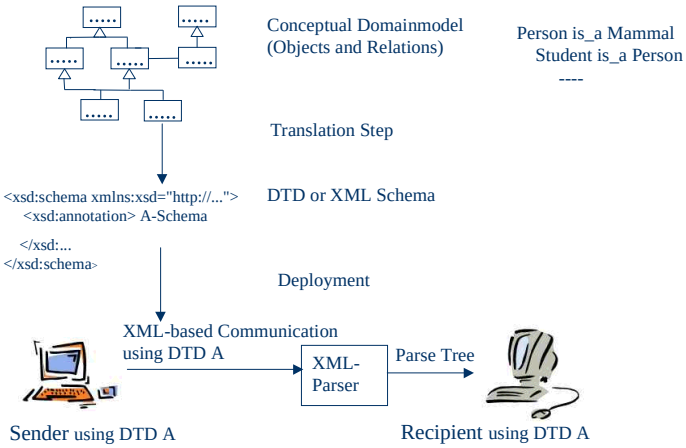
```
<Creator>
    <uri> http://www.w3.org/Home/Lassila </uri>
    <name>Ora Lassila</name>
</Creator>
```

```
<Document uri="http://www.w3.org/Home/Lassila"
    <Creator>Ora Lassila</Creator>
</Document>
```

```
<Document uri="http://www.w3.org/Home/Lassila" Creator="Ora Lassila"/>
```

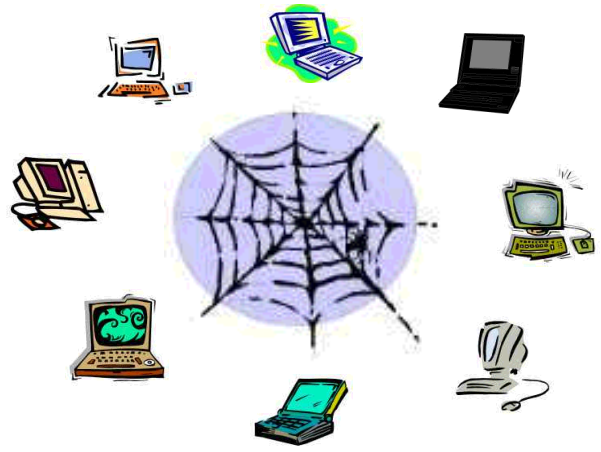
RDF

Point to Point Communication for Machine-Understandable Data



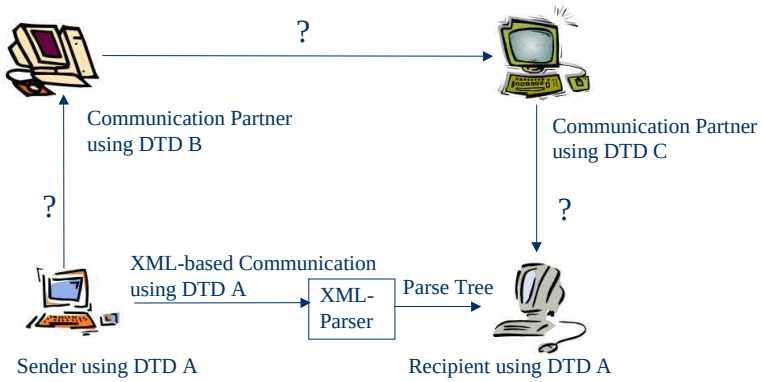
RDF

Many Previously Unknown Communication Partners



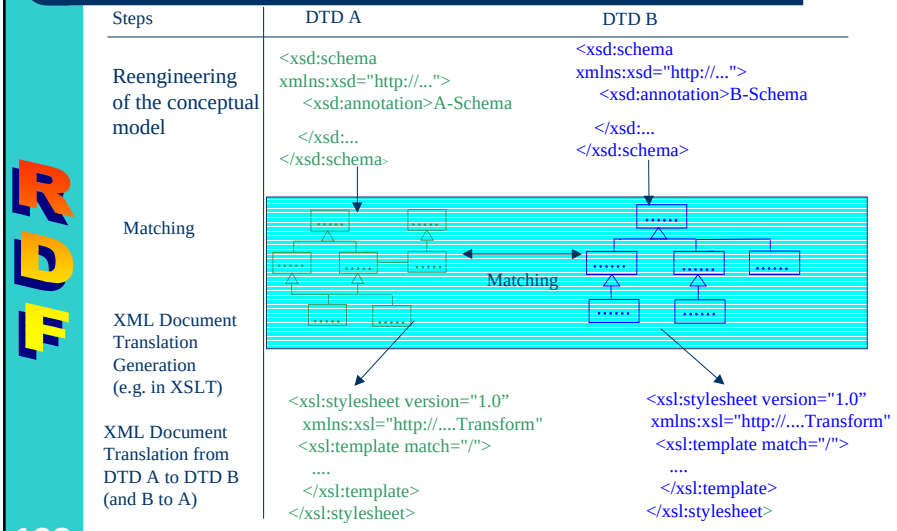
RDF

New Partners Don't Understand Each Other



RDF

Merging Steps Between Models



Merging/Aligning Models

- Reengineering step is costly and unnecessary, when a conceptual language is in use
- Generation document translation procedures is again complicated and unnecessary

==> use a level on top of XML

- What are requirements for such a level?

Postulates: Fundamental Requirements for KR on the Web

1. Knowledge on the Web is distributed
   (link Knowledge on the Web)
2. Knowledge on the Web is biased - there is no universal truth
   it must be possible to dispute statements
3. Many different user communities:
   Extensibility and Simplicity

==> Resource Description Framework (RDF)

Introduction to RDF

- RDF (Resource Description Framework)
  - Beyond Machine readable to *Machine understandable*
- RDF unites a wide variety of stakeholders:
  - Digital librarians, content-raters, privacy advocates, B2B industries, AI...
  - Significant (but less than XML) industrial momentum, lead by W3C
- RDF consists of two parts
  - RDF Model (a set of triples)
  - RDF Syntax (different XML serialization syntaxes)
- RDF Schema for definition of Vocabularies (simple Ontologies) for RDF (and in RDF)

RDF Data Model

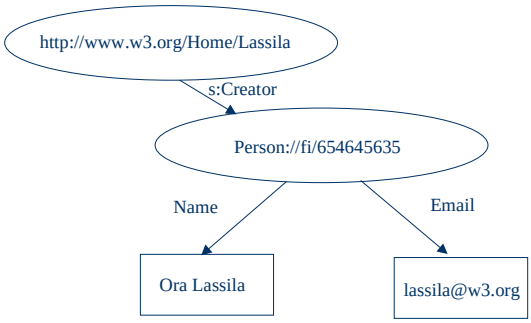
- **Resources**
  - A resource is a thing you talk about (can reference)
  - Resources have URI's
  - RDF definitions are itself Resources (linkage, see requirement 1)
- **Properties**
  - slots, defines relationship to other resources or atomic values
- **Statements**
  - "Resource has Property with Value"
  - (Values can be resources or atomic XML data)
- **Similar to Frame Systems**

A Simple Example

- **Statement**
  - "Ora Lassila is the creator of the resource http://www.w3.org/Home/Lassila"
- **Structure**
  - Resource (subject) http://www.w3.org/Home/Lassila
  - Property (predicate) http://www.schema.org/#Creator
  - Value (object) "Ora Lassila"
- **Directed graph**

Another Example

- To add properties to Creator, point through a intermediate Resource.

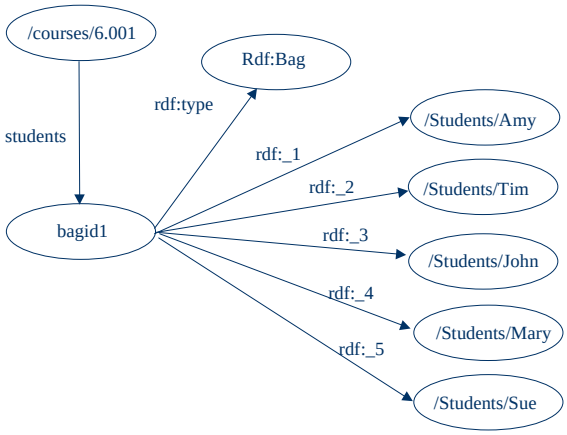


Collection Containers

- Multiple occurrences of the same PropertyType doesn't establish a relation between the values
  - The Millers own a boat, a bike, and a TV set
  - The Millers need (a car or a truck)
  - (Sarah and Bob) bought a new car
- RDF defines three special Resources:
  - **Bag** unordered values rdf:Bag
  - **Sequence** ordered values rdf:Seq
  - **Alternative** single value rdf:Alt
    - Core RDF does not enforce 'set' semantics amongst values

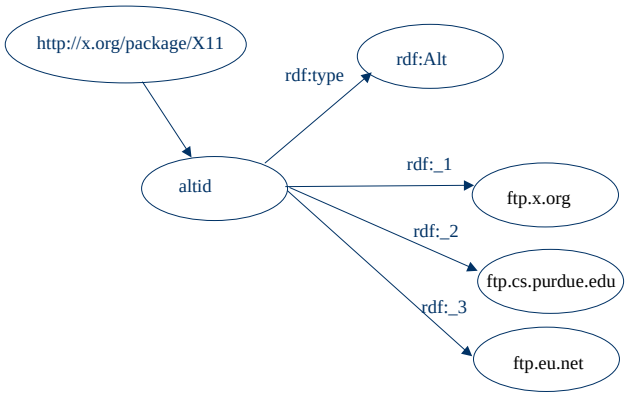
Example: **Bag**

- *The students in course 6.001 are Amy, Tim, John, Mary, and Sue*



Example: **Alternative**

- *The source code for X11 may be found at ftp.x.org, ftp.cs.purdue.edu, or ftp.eu.net*

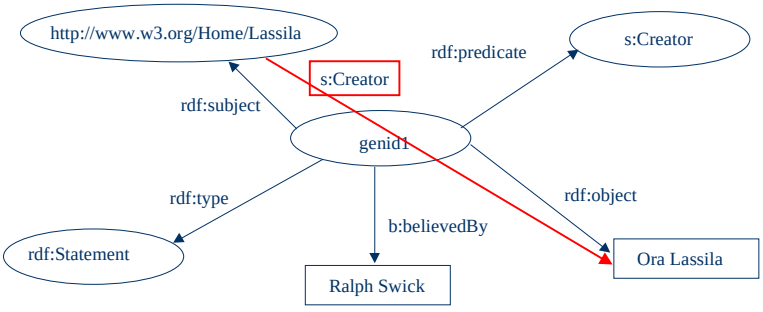


Statements About Statements (Requirement 2: Dispute Statements)

- Making statements about *statements* requires a process for transforming them into Resources
  - **subject** the original referent
  - **predicate** the original property type
  - **object** the original value
  - **type** rdf:Statement

Example: **Reification**

- *Ralph Swick believes that*
  - *the creator of the resource http://www.w3.org/Home/Lassila is Ora Lassila*



A Formal Model of RDF

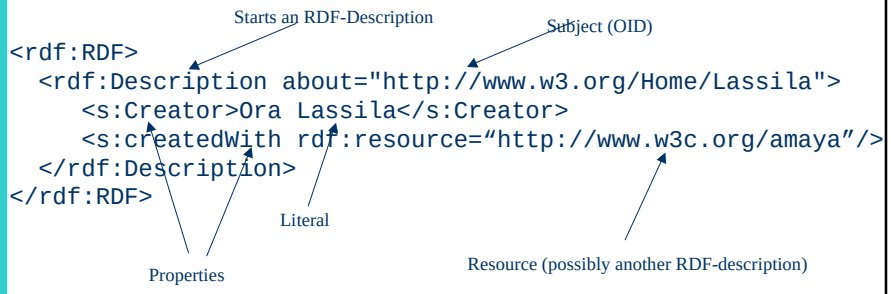
- RDF itself is mathematically straightforward:
  - **Basic Definitions**
    - Resources.
    - Properties $\subset$ Resources
    - Literals
    - Statements = Resources $\times$ Properties $\times$ {Resources $\cup$ Literals}
  - **Typing**
    - rdf:type $\in$ Properties
    - {RDF:type, sub, obj} $\in$ Statements $\Rightarrow$ obj $\in$ Resources

Formal Model of RDF II

- **Reification**
  - rdf:Statement $\in$ Resource-Properties
  - {rdf:predicate, rdf:subject, rdf:object} $\subset$ Properties
  - Reification of a triple { pred, sub, obj } of Statements is an element r of Resources representing the reified triple and the elements s1, s2, s3, and s4 of Statements such that
    - s1: {RDF:predicate, r, pred}
    - s2: {RDF:subject, r, subj}
    - s3: {RDF:object, r, obj}
    - s4: {RDF:type, r, [RDF:Statement]}
- **Collections**
  - { RDF:Seq, RDF:Bag, and RDF:Alt } $\subset$ Resources-Properties
  - There is a subset of Properties corresponding to the ordinals (1, 2, 3, ...) called Ord. We refer to
  - elements of Ord as RDF:_1, RDF:_2, RDF:_3, ...

RDF Syntax I

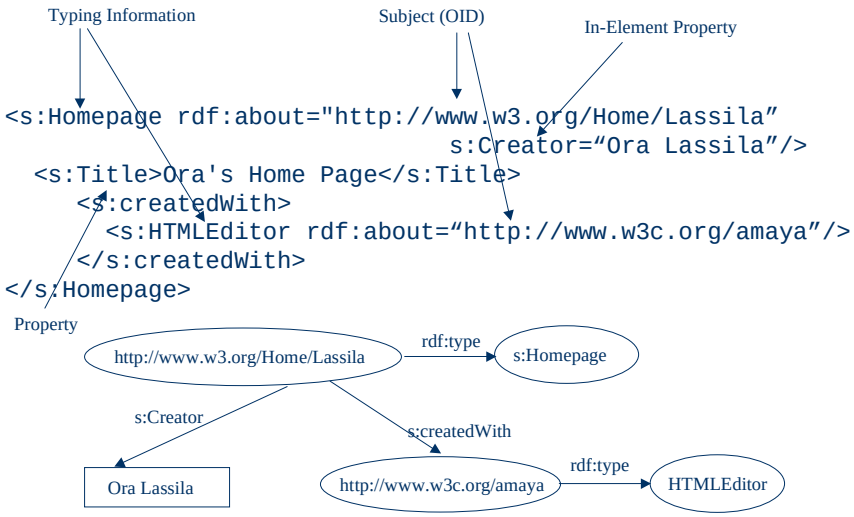
- Data model does not enforce particular syntax
- Specification suggests many different syntaxes based on XML
- General form:



Resulting Graph


```
<rdf:RDF>
  <rdf:Description about="http://www.w3.org/Home/Lassila">
     <s:Creator>Ora Lassila</s:Creator>
     <s:createdWith rdf:resource="http://www.w3c.org/amaya"/>
  </rdf:Description>
</rdf:RDF>
```

RDF Syntax II: Syntactic Varieties



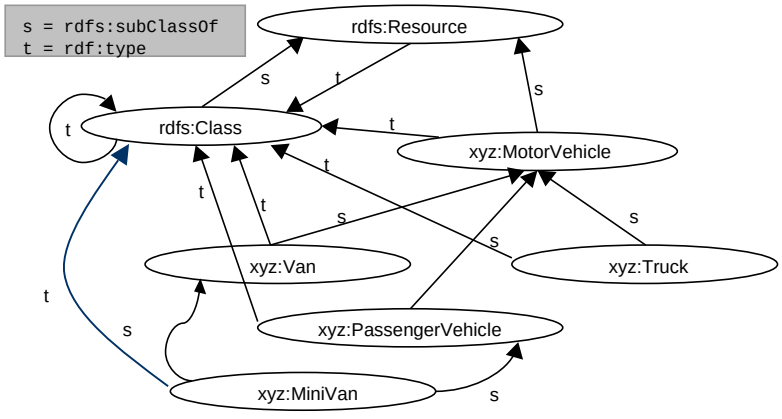
RDF Schema (RDFS)

- RDF just defines the data model
- Need for definition of vocabularies for the data model - an Ontology Language!
- RDF schemas are Web resources (and have URIs) and can be described using RDF

Most Important Modeling Primitives

- Core Classes
  - Root-Class `rdfs:Resource`
  - MetaClass `rdfs:Class`
  - Literals `rdfs:Literal`
- `rdfs:subClassOf`-property
- Inherited from RDF: properties (slots)
- `rdfs:domain` & `rdfs:range`
- `rdfs:label`, `rdfs:comment`, etc.
- Inherited from RDF: InstanceOf (`rdf:type`)

RDF-Schema: Example



Example: RDF-Schema in RDF-Schema

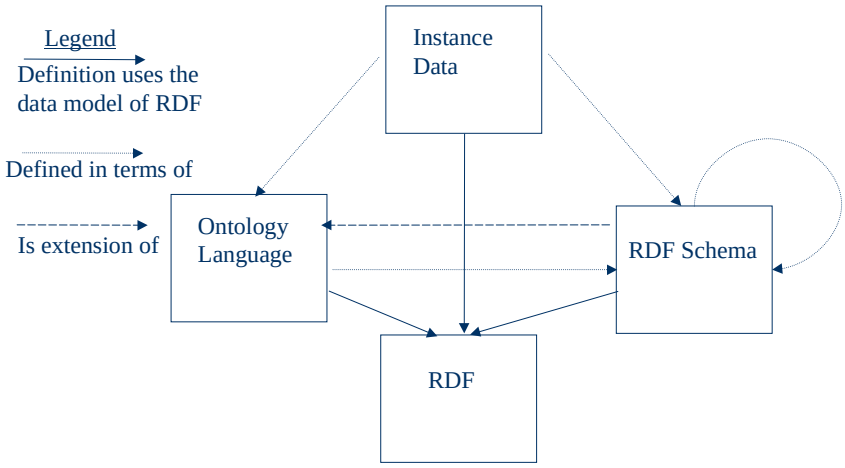
- Namespace-URL: http://www.w3.org/2000/01/rdf-schema#

```
<rdfs:Class rdf:ID="Resource">
  <rdfs:label xml:lang="en">Resource</rdfs:label>
  <rdfs:label xml:lang="fr">Ressource</rdfs:label>
  <rdfs:comment>The most general class</rdfs:comment>
</rdfs:Class>
<rdfs:Class rdf:ID="Class">
  <rdfs:label xml:lang="en">Class</rdfs:label>
  <rdfs:label xml:lang="fr">Classe</rdfs:label>
  <rdfs:comment>The concept of Class</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>
<rdf:Property ID="subClassOf">
  <rdfs:label xml:lang="en">subClassOf</rdfs:label>
  <rdfs:label xml:lang="fr">sousClasseDe</rdfs:label>
  <rdfs:comment>Indicates membership of a class</rdfs:comment>
  <rdfs:range rdf:resource="#Class"/>
  <rdfs:domain rdf:resource="#Class"/>
</rdf:Property>
```


Extensibility of RDF

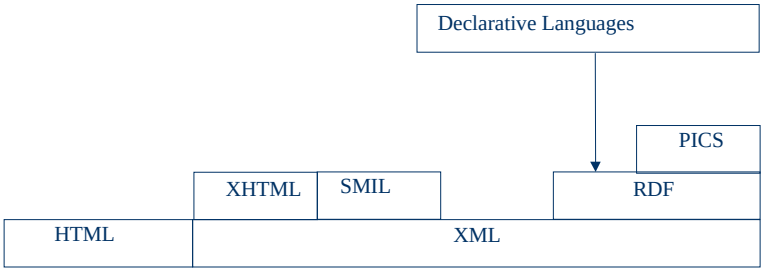
- Define an Ontology of your Language with RDF Schema (like RDF-Schema itself)
- Describe Instance Data using your new Vocabulary
- Advantage: all Languages use the same Data Model (simplifies Interoperability)

Ontology Languages on Top of RDF: The Principle

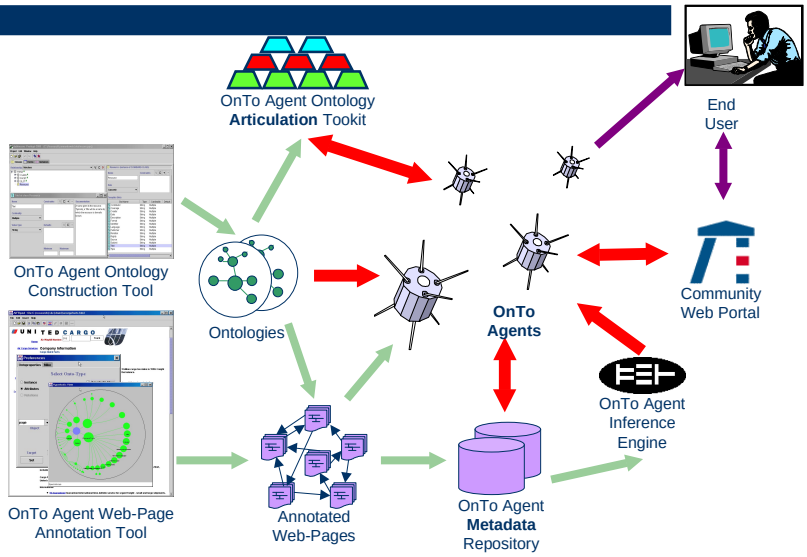


The Semantic Web

- A Web of machine understandable Data, based on declarative languages on top of RDF (all use the same data model!)
- Intelligent Agent enabling architecture
- W3C's vision for the Semantic Web Architecture:



DARPA Agent Markup Language (DAML): An Information Foodchain Based on RDF



RDF-Resources

- RDF-Editor: Protege
  www-smi.stanford.edu/projects/protege
- RDF-Parser and APIs/Query Engines
  - http://www-db.stanford.edu/~melnik/rdf
  - http://www.aifb.uni-karlsruhe.de/~sde/rdf
- RDF Knowledge Sources
  - DMOZ - Open Directory (largest human created Web-directory) http://www.dmoz.org
- General Information:
  - RDF Interest Mailing list: `www-rdf-interest@w3.org`
    Archive: `http//lists.w3.org/Archives/Public/www-rdf-interest/`
  - **`SemanticWeb.org`**

Open Issues of RDF

- RDF-Schema is still evolving
- URIs: What is the URI of a person? Other real world entity?
- “Killer Applications”: Search Engines? B2B-Communication (e.g. ebXML.org ?)
- Interoperability between RDF-Schemas

Acknowledgements

- William Grosso, Rohit Khare, Peter Karp, Dan Brickley for their kind permission to use material
- Andreas Abecker, Heiko Maus, Andreas Lauer, Hansi Hein, Stefan Klink, Bertin Klein, and other colleagues at DFKI for fruitful discussions
- Peter Fankhauser for valuable input on query languages
- Elliotte Rusty Harold for his 100% comprehensive XML Bible: http://metalab.unc.edu/xml/books/bible/

References

References

Cocoon
The Cocoon Publishing Framework. Available at http://xml.apache.org/cocoon/index.html

CSS2
Cascading Style Sheets, level 2. Available at http://www.w3.org/TR/REC-CSS2/

DAML
DARPA Agent Markup Language. Availabe at http://www.oasis-open.org/cover/daml.html , http://dtsn.darpa.mil/iso/programtemp.asp?mode=347 , and http://www.daml.org/

Dublin Core
Available at http://purl.oclc.org/dc/

Fernandez99
XML Query Languages: Experiences and Exemplars, Mary Fernandez, Jérôme Siméon, Philip Wadler, 1999. Available at http://www.w3.org/1999/09/ql/docs/xquery.html

FRODO
The FRODO Project. Homepage available at http://www.dfki.uni-kl.de/frodo/

OIL
Ontology Interchange Language. Availabe at http://www.ontoknowledge.org/oil/

Protégé-XML
The Protégé XML-Tab. Available at http://smi-web.stanford.edu/people/sintek/

References

Q&T
Querying and Transforming XML. Available at http://www.w3.org/TandS/QL/QL98/pp/query-transform.html

RDF
Homepage: http://www.w3.org/RDF/
Specification: http://www.w3.org/TR/REC-rdf-syntax
FAQ: http://www.w3.org/RDF/FAQ
Tutorial: Carl Lagoze et al: WWW7 Tutorial. Available at http://purl.oclc.org/~emiller/talks/www7/tutorial

RDF-Schema
Specification at http://www.w3.org/TR/rdf-schema

RFML
The Relational-Functional Markup Language. Homepage at http://www.relfun.org/rfml/

Semantic Web
Availabe at http://www.SemanticWeb.org

SHOE
Simple HTML Ontology Extensions. Available at http://www.cs.umd.edu/projects/plus/SHOE/

XML
Extensible Markup Language (XML), Version 1.0. W3C Recommendation. Available at http://www.w3.org/TR/1998/REC-xml-19980210

References

XML Namespaces
Namespaces in XML. W3C Recommendation at http://www.w3.org/TR/REC-xml-names/

XML-QL
XML-QL: A Query Language for XML. Available at http://www.w3.org/TR/NOTE-xml-ql/

XML-RPC
XML Remote Procedure Calls. Available at http://www.xmlrpc.com/

XPath
World Wide Web Consortium. XML Path Language. W3C Recommendation. Available at http://www.w3.org/TR/xpath

XPointer
World Wide Web Consortium. XML Pointer Language (XPointer). W3C Working Draft. Available at http://www.w3.org/TR/xptr

XQL
XML Query Language (XQL). Available at http://www.w3.org/TandS/QL/QL98/pp/xql.html

XSL
World Wide Web Consortium. Extensible Stylesheet Language (XSL). W3C Working Draft. Available at http://www.w3.org/TR/WD-xsl

XSLT
World Wide Web Consortium. XSL Transformations (XSLT). W3C Recommendation. Available at http://www.w3.org/TR/xslt