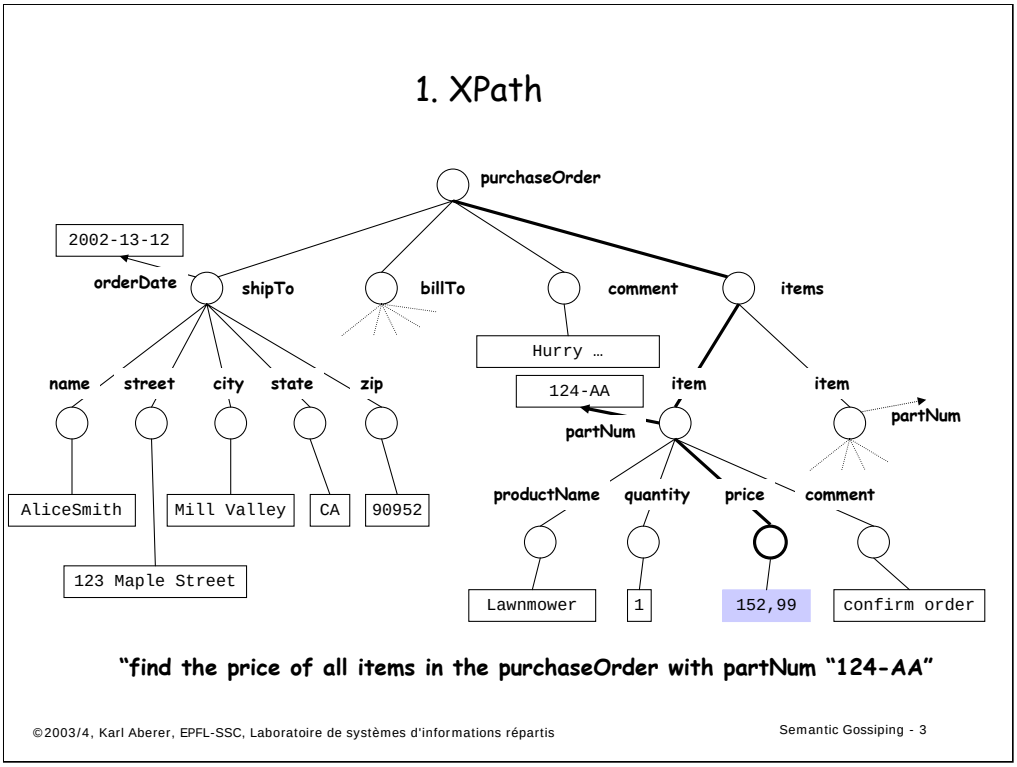
Special Session:
Semantic Gossiping

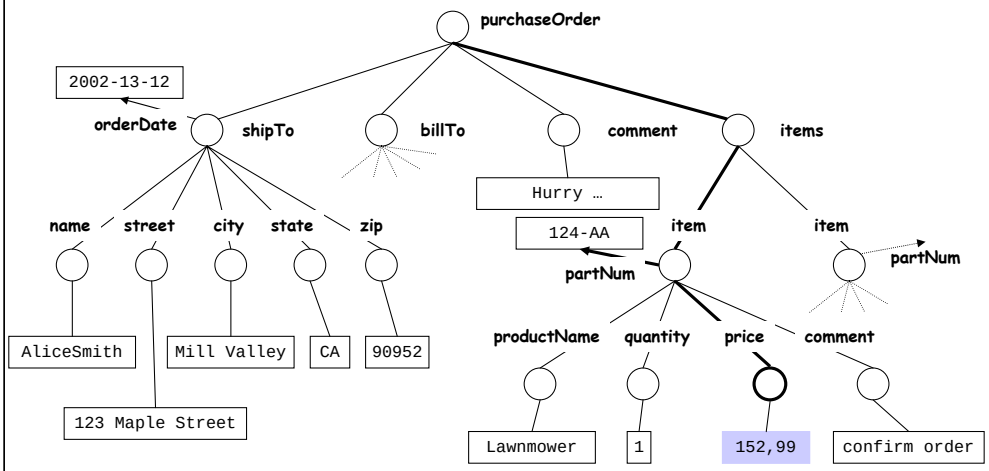
Overview

- 1. XPath revisited
- 2. XQuery
- 3. XSLT
- 4. Introduction to Semantic Gossiping
- Exercise / lab on Semantic Gossiping



The problem to address is the following: given an XML document, how can we answer queries as the following: "find the price of all items in the purchaseOrder with partNum "124-AA". For relational databases this would be a typical query to be expressed in SQL. We introduce now the corresponding counterparts for XML.

Example XPath Query



/purchaseOrder/items/item[@partnum="124-AA]/price

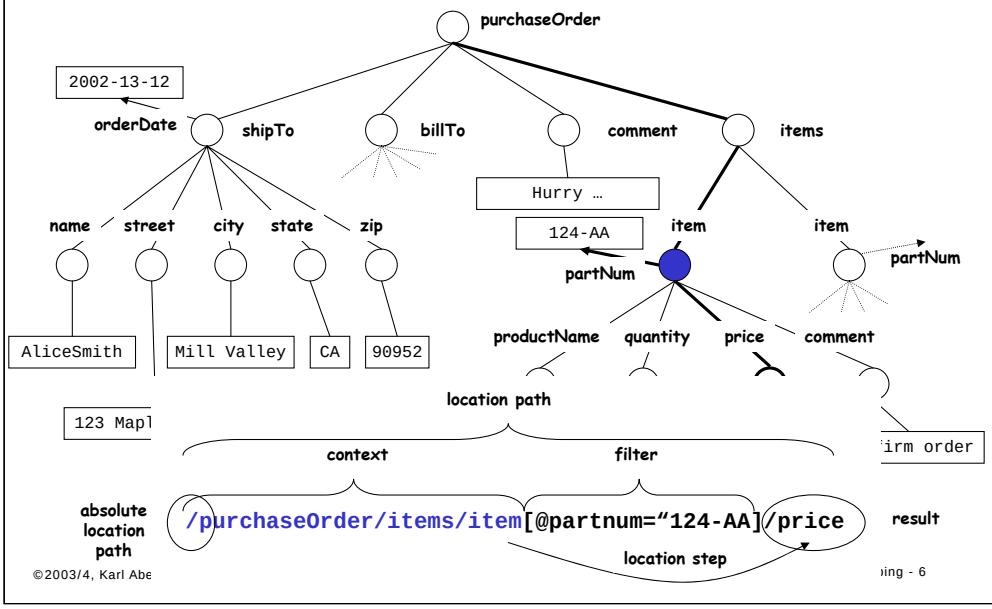
XPath Overview

- Logical adressing of document parts
- Key concepts
  - Location paths select nodes relative to a given context node
  - Absolute location paths start at the document root
  - Location paths consist of a sequence of location steps
  - The last location step determines the result
  - Filter expressions contain location paths
  - Context of filter expressions is the associated location step
- Practical aspects
  - Non-XML Syntax
  - Used by other XML standards (XSLT, Xpointer)
  - Used within XML attributes and URIs

When considering XML documents as data, we need to provide a query language to access this data. As the data view is a tree representation, a first capability of any XML query language is to navigate along the paths in the tree. This is the essential function that Xpath provides. It allows to address (specify) document parts by providing a navigation « instruction » how the part can be reached. The basic principle of Xpath is to navigate in tree document in a manner comparable to the navigation in a directory tree in a file system and to evaluate at each step additional filter conditions. The navigation steps (location steps) and the evaluation of filter conditions depend on the current navigation context (the place in the tree where the navigation has arrived). It is important to understand that the result of an Xpath query is always a set of element nodes (and nothing else, in particular not a XML document fragment)

From a practical viewpoint it is interesting to mention that Xpath does not use an XML syntax, thus an Xpath expression is not a well-formed XML document (other standards, e.g. XSLT, the XML document transformation language, use XML syntax to denote expressions in their specific model). Xpath is a component that is reused in other standards, notably in XSLT. Xpath expressions are also intended to extend URLs to URIs (universal resource identifier) to address parts of XML documents.

Example XPath Query



XPath Location Paths

- A location step consists of
  - an axis (the navigation direction),
  - a node test, and
  - a predicate
- Axis operators
  - `AxisName ::= 'ancestor' | 'ancestor-or-self' | 'attribute' | 'child' | 'descendant' | 'descendant-or-self' | 'following'| 'following-sibling' | 'namespace' | 'parent' | 'preceding' | 'preceding-sibling' | 'self'`
  - ! Preceding axis exclude any ancestor node, following axis exclude any descendant node
- Example: absolute location path

```
/child::purchaseorder[child::shipto/child::name="Alice"]/
     child::items/child::item[position()=1]
```

- Abbreviated syntax (used in practice)

```
/purchaseorder[shipto/name="Alice"]/items/item[1]
```


The simple navigation pattern in the previous example (downwards to elements with specific names) is generalized in Xpath. Each navigation step is characterized by three things:

The direction, i.e. a navigation needs not necessarily move from a parent to a child node, but can follow any relation among elements, in particular by traversal of elements at the same tree level according to the document order (following, preceding) and traversal of multiple nodes (descendants). A complete account of the possible navigation operators is given. The node test checks whether a element node that is encountered in the navigation matches a certain element name. And finally the predicates are the filter expressions which allow to select nodes based on other properties than their name. In particular, filters allow to use other Xpath queries to check a property of an element. In that case the predicate is considered as successfully evaluated if a non-empty result set is generated. In order to take account of the different axis operators an extended syntax is used in Xpath that specifies for each location step the axis operator. In practice however, the abbreviated syntax that we have already seen is more common.

XPath Abbreviated Syntax

- Selection of elements
  - `item`
    - selects from the current context all elements with name `item`
- Hierarchy operators
  - `item/price`
    - all `price` elements child of `item`
  - `./item`
    - equivalent to `item`
  - `purchaseOrder//item`
    - all descending elements with name `item`
  - `name/..`
    - parent node of `name`

We introduce all operators of the abbreviated syntax by means of examples. The hierarchy operators are straightforward to understand as their semantics coincides with the UNIX directory navigation semantics, except the // operator. It represents the navigation to all descendent elements of the current context element, i.e. all elements on the paths to the leafs are selected (and not only the leaf nodes).

XPath Abbreviated Syntax

- Wildcards
  - `purchaseOrder/*/item`
    - all `item` that are reachable by passing through one arbitrary element
  - `*/*`
    - all grandchildren
  - `@*`
    - all attributes
- Indexed Access
  - `item[1]`
    - first `item` element
  - `item[1, 4]`
    - first and fourth `item` element
  - `item[last()]`
    - last `item` element

Wildcards match all element names, in other words no node test is performed. Be careful: the wildcard * has in Xpath another meaning than in a regular expression or in an XML DTD, where it indicates repeated occurrence, which is covered in turn in Xpath by the // operator. Also attributes can be selected. This is particularly useful in filter expressions when attribute values need to be checked.

Indexed access allows to traverse neighbouring elements at the same tree level. We see here also one example of using a (built-in) function in a predicate, namely last(). A number of basic functions for node access and string manipulation are specified in Xpath.

XPath Abbreviated Syntax

- Filter
  - `item[price]`
    - all `item` elements containing a `price` element
  - `purchaseOrder[billTo]/items[item]`
    - All `items` elements containing an `item` element that are contained in a `purchaseOrder` element containing a `billTo` element
  - `purchaseOrder[items/item]`
  - `purchaseOrder[shipTo and billTo]`
  - `item[productName="car"]`
    - All `item` elements containing a `productName` element with textual content « `car` »
  - `item[@partNumber="A100"]`

- Union
  - `purchaseOrder/billTo/name | purchaseOrder/shipTo/name`
  - Only at top level !

We see that two types of filter expressions exist. One that returns an element (or attribute) set. These are true if the sets are non-empty. In this way also Boolean combinations of Xpath expressions (e.g. shipTo and billTo) make sense. And others returning a Boolean value (e.g. using the equality predicate). These are true if for at least one element in the Xpath expression appearing in the predicate the condition is satisfied. Xpath provides only one set operator for set union (in contrast to SQL) and this with the restriction that it can only be applied at the top level of the Xpath expression.

XPath Abbreviated Syntax Operators

/	Child operator	document element that is the direct child
//	Recursive descent	all elements in the document that are indirect child
.	Current context node	
*	Wildcard	matches all element and attribute names
@	Attribute	distinguishes attributes from elements
[]	Filter	
f( )	Method call	
( )	Grouping	

2. XQuery - Querying XML Data

- Problem: XPath lacks basic capabilities of database query languages, in particular join capability and creation of new XML structures
- XQuery extends XPath to remedy this problem
- Additional concepts in XQuery
  - Extended path expressions
  - Element constructors
  - FLWR expressions
  - Expressions involving operators and functions
  - Conditional expressions
  - Quantified expressions

By now it should have become clear that Xpath lacks basic capabilities one would expect from a (declarative, set-oriented) database querying language. In particular it has no general support for set operators, it allows to return only element and attribute sets and is thus not closed and it has no support of an operation that is equivalent to a relational join, which would be required to establish value-based relationships among XML document parts. We introduce now the most important additional concepts of XQuery as they were specified in June 2001. The goal is not to obtain a thorough knowledge and capability to use Xquery, but to understand the substantial additional concepts to Xpath and the relationships to SQL. With a knowledge of Xpath and SQL the following presentation of Xquery should be straightforward to follow.

Dereference Operator

- `document("zoo.xml")//`
  `chapter[title = "Frogs"]//figref/@refid->fig/caption`
  - Find captions of figures that are referenced by `figref` elements in the chapter of "`zoo.xml`" with title "`Frogs`".

```
document.xml:
<chapter>
  <title>Apples<\title>
  <para>
    <fig id="1">
     <caption> this is a figure<\caption>
    <\fig>
  <para>
<\chapter>
<chapter>
  <title>Frogs<\title>
    <references>
      <figref refid="1"\>
    <\references>
<\chapter>
```


Navigation in Xquery is possible over IDREF attributes. In this example first the figref element would be located from which the IDREF value is taken and the fig element with that ID value is located, leading to the result.

The expression document(« zoo.xml ») replaces the / operator.

Element Constructor and FLWR Expressions

```
<result>
{
   LET $a := avg(document("bib.xml")//book/price)
   FOR $b IN document("bib.xml")//book
   WHERE $b/price > $a
   RETURN
       <expensive_book>
          {$b/title}
          <price_difference>
             {$b/price - $a}
          </price_difference>
       </expensive_book>
}
</result>
```

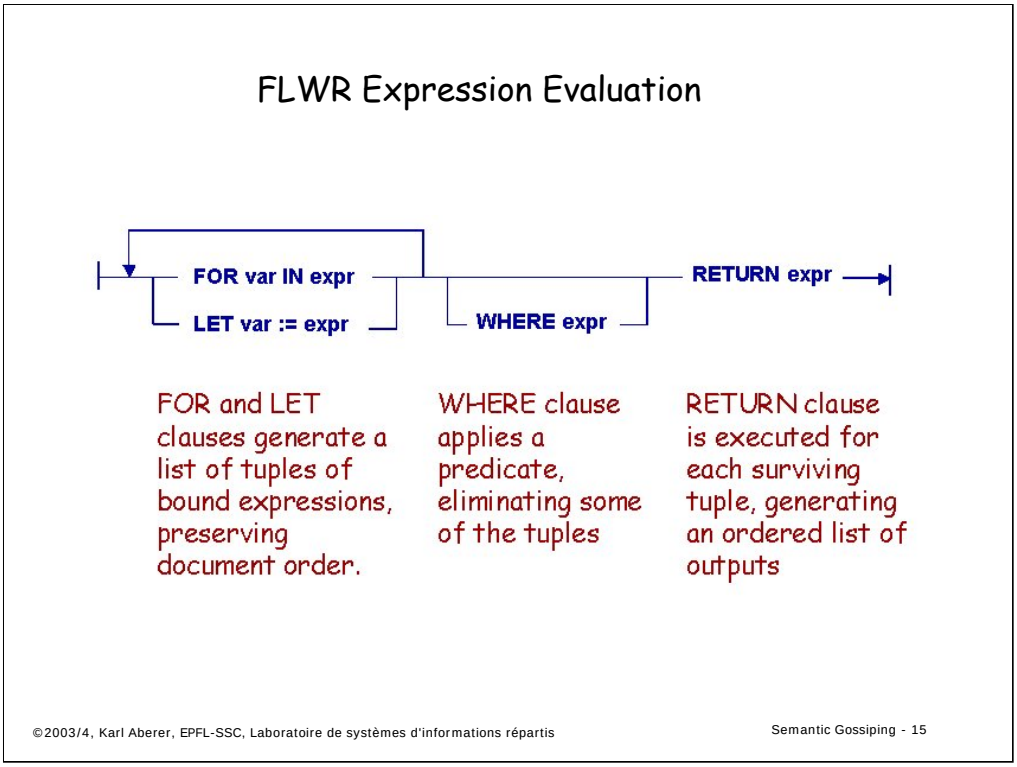
"macro"

"FROM"

"SELECT"

- For each book whose price is greater than the average price, return the title of the book and the amount by which the book's price exceeds the average price

This query exhibits a whole wealth of new concepts of Xquery. Most notably one can see that Xquery allows variable binding as SQL. Different to SQL where relations are available to bind variables in Xquery they have to bind to sets that result from other Xquery expressions (this is the only possibility to obtain sets). This is done in the FOR clause. In addition using the LET clause one can introduce variables that factor out repeatedly occurring expressions in the queries. Note that these variables are used very differently from the ones bound to set valued expressions: they are just syntactically replaced in the query. The second observation is that a WHERE clause is available to express conditions. This allows in particular to express joins when multiple variables are bound in the FOR clause. The third observation is that a RETURN clause allows to return structured results, creating new XML document fragments. Finally we see that a query expression itself can be nested within a XML document fragment.



The semantics of Xquery expressions is defined similarly to SQL (which is in short: build the Cartesian product of the relations in the FROM clause, evaluate the predicates in the WHERE clause and then project on the attributes in the SELECT clause). Also for FLWR expression first generate all tuples from the Cartesian product space of all sets to which variables are bound. An important difference that the order among document elements needs to be preserved, therefore also the order in which the variables appear in the FOR clause has an impact on the order the result tuples will be sorted. The WHERE clause is evaluated as for SQL and for each remaining tuple an XML document fragment is generated by replacing the variables by the tuple values. There is also a XML query algebra under development which is intended to provide a precise semantics to Xquery.

Join Example

```
<result>
{
FOR $seller IN document("users.xml")//user_tuple,
    $buyer IN document("users.xml")//user_tuple,
    $item IN document("items.xml")//item_tuple,
    $highbid IN document("bids.xml")//bid_tuple
WHERE
    $seller/name = "Tom Jones" AND
    $seller/userid = $item/offered_by AND
    contains($item/description, "Bicycle") AND
    $item/itemno = $highbid/itemno AND
    $highbid/userid = $buyer/userid AND
    $highbid/bid = max(document("bids.xml")//bid_tuple
           [itemno = $item/itemno]/bid)
RETURN
    <jones_bike>
    { $item/itemno } { $item/description }
           <high_bid> { $highbid/bid } </high_bid>
           <high_bidder> { $buyer/name } </high_bidder>
    </jones_bike>
SORTBY(itemno) }
</result>
```

assumes an XML representation of a relational database

Functions

- NAMESPACE
  xsd=http://www.w3.org/2001/03/XMLSchema-datatypes

```
FUNCTION depth(ELEMENT $e) RETURNS xsd:integer
{   -- An empty element has depth 1
    -- Otherwise, add 1 to max depth of children
  IF empty($e/*) THEN 1
  ELSE max(depth($e/*)) + 1 }

depth(document("partlist.xml"))
```

  - Find the maximum depth of the document named "partlist.xml."

Xquery goes much further in terms of expressiveness than SQL by allowing the definition of arbitrary user-defined functions. This is a feature that can be found in SQL99 (as well), the object-relational query language standard building on SQL92. We also see in this example, that Xquery uses type specifications that are provided in the XML Schema standard.

Existential and Universal Quantifiers

- ```
  FOR $b IN //book
  WHERE
     SOME $p IN $b//para SATISFIES contains($p, "sailing")
     AND contains($p, "windsurfing")
  RETURN $b/title
  ```
  - Find titles of books in which both sailing and windsurfing are mentioned in the same paragraph.
- ```
  FOR $b IN //book
  WHERE
    EVERY $p IN $b//para SATISFIES contains($p, "sailing")
  RETURN $b/title
  ```
  - Find titles of books in which sailing is mentioned in every paragraph.

As in SQL, Xquery also supports the concepts of universal and existential quantification of variables ranging over set expressions.

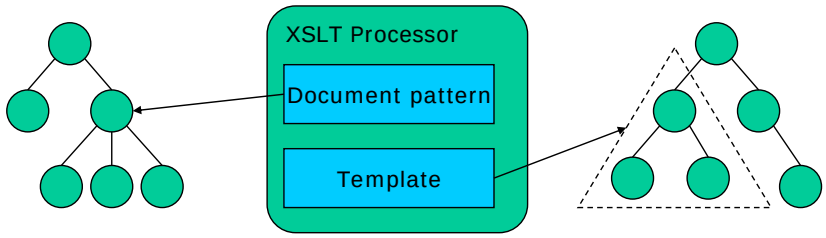
XQuery Development

- Conciliates many approaches
  - Path expressions from XPath
  - Variable binding concept from XML-QL
  - SELECT-FROM-WHERE paradigm from SQL92
  - Orthogonality from OQL
  - User-defined functions from SQL99
- Semantics given in terms of the XML query algebra

We have seen that Xquery unifies concepts from many different origins. The variable binding and result generation concept used was in fact for the first time specified in a research system, XML-QL. Most of the other ideas are borrowed from the relational and object-relational query languages. In particular the orthogonality reminds a lot of OQL (Object Query Language): orthogonality means that everywhere in an expression where it is type-safe, an arbitrary expression of the language can be substituted.

3. XSLT

- Restructuring of XML documents
  - Layout of documents
  - overcoming heterogeneity, transformation of schemas



Source document + Rule set (XSLT Stylesheet) = Target document

XSLT Example

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

<chapter>First Chapter</chapter>

<chapter>Second Chapter
 <chapter>Subchapter 1</chapter>
 <chapter>Subchapter 2</chapter>
</chapter>

<chapter>Third Chapter
 <chapter>Subchapter A</chapter>

 <chapter>Subchapter B
  <chapter>sub a</chapter>
  <chapter>sub b</chapter>
 </chapter>

 <chapter>Subchapter C</chapter>
</chapter>
</LectureNotes>
```

XML Source

```
<xsl:stylesheet
 xmlns:xsl=
 'http://www.w3.org/XSL/Transform/1.0'
>
 <xsl:template match="/">
  <TABLE BORDER="1">
   <TR>
    <TH>Number</TH>
    <TH>text</TH>
   </TR>
   <xsl:for each select="//chapter">
    <TR>
     <TD><xsl:number/></TD>
     <TD>
       <xsl:value- of
        select="./text()"
       />
     </TD>
    </TR>
   </xsl:for each>
  </TABLE>
 </xsl:template>
</xsl:stylesheet>
```

XSLT Stylesheet

Number	text
1	First Chapter
2	Second Chapter
1	Subchapter 1
2	Subchapter 2
3	Third Chapter
1	Subchapter A
2	Subchapter B
1	sub a
2	sub b
3	Subchapter C

HTML Output (formatted)

XSLT Concepts

- Declarative programming language
- Functional programming paradigm (no side-effects!)
- Xpath is used as locator language
- XML representation of programs
- Programming constructs
  - Loops, conditional statements, sorting
  - Templates (correspond to parametrized functions)
  - Pattern matching (for invocation of templates)
  - Copying and creation of document constituents (elements, attributes, text,…)
- Program execution
  - Input document is processed in a top-down fashion
  - Output is generated during processing
  - Processing context: each command is executed in a context consisting of a node set, initial context is the root node
  - Conflict resolution for templates
  - Default processing rules

Two Types of XSLT Stylesheets

- Literal result-element generation
  - Instantiates the document by processing the embedded XSLT statements

```
<?xml version='1.0' ?>
<doc xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
xsl:version='1.0'>
<xsl:copy-of select='/publications//author'/>
</doc>
```

- Template-based stylesheet
  - Processes the templates starting from the document root

```
<?xml version='1.0' ?>
<xsl:stylesheet version='1.0'
xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:template match='//paper' >
    <contributor> <xsl:value-of select='authors/author' />
    </contributor>
</xsl:template>
</xsl:stylesheet>
```

Basic XSLT Programming Constructs: Generating Output

- Literal result element generation
- Copying from the input document
  - <xsl:copy-of select=' '>: copies all nodes from the node set selected (and their subelements)
  - <xsl:value-of select=' '>: copies the textual content of the first node of the node set selected

```
<result><xsl:copy-of select='//authors'/></result>
```

```
<authors><author>W. Klas</author><author>G. Fischer</author><author>K.
Aberer</author></authors>
<authors><author>M. Volz</author><author>K. Aberer</author><author>K.
Böhm</author></authors>
```

```
<result><xsl:value-of select='//authors'/></result>
```

```
<result>W. Klas G. Fischer K. Aberer</result>
```

- Explicit instantiation of nodes

```
<result>
<xsl:element name='doc'/>
<doc>
<xsl:attribute name='id' >32</xsl:attribute></doc></result>
```

```
<result version='1.0'><doc/><doc id='32'/></result>
```

Basic XSLT Programming Constructs: Program Logic

- Conditional statement: <xsl:if test=' '>
- For-each loop: <xsl:for-each select=' '>
- Sorting of nodes in for-each: <xsl:sort select=' '/>
- Example

```
<authorcount
xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
xsl:version='1.0'>
<xsl:for-each select='//paper'>
 <xsl:sort select='year'/>
  <xsl:if test='count(.//author) > 2 '>
    <xsl:copy-of select='title'/>
    <nr><xsl:value-of select='count(.//author)'/></nr>
  </xsl:if>
</xsl:for-each>
</authorcount>
```

XSLT Templates

- XSLT templates correspond to functions
  - Allow for modularization of stylesheets
  - Allow recursive programming !
  - Can be called with parameters
- Invocation
  - Explicitely by a given name

```
<xsl:template name='name' >  called by
<xsl:call-template name='name'/>
```

  - Implicitely by pattern matching (declarative programming)
  - Pattern given as Xpath expression

```
<xsl:template match='pattern'>  called by
<xsl:apply-templates select=' '/>
```

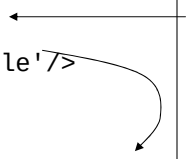
Example

```
<?xml version='1.0' ?>
<xsl:stylesheet version='1.0'
xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>

<xsl:template match='//paper'>
 <xsl:call-template name='printTitle'/>
</xsl:template>

<xsl:template name='printTitle' >
  <TITLE><xsl:value-of select='title'/></TITLE>
</xsl:template>

</xsl:stylesheet>
```



Invocation for each element of context

Template Processing

- Initialisation
  - Implicitely an xsl:apply-templates statement is applied at the beginning with the root node as context

```
<xsl:template match='/' >
    <xsl:apply-templates/>
</xsl:template>
```

- Conflict resolution
  - Only templates that match the current context node may be chosen
  - Imported and included rules have lower priority
  - Otherwise the rule with higher priority is chosen
    - Priority is computed from structure of matching template or explicitely given
  - Two equal priority rules are an error
- Default templates
  - If no template applies to a node in a context that needs to be processed due to an apply-template statement, built-in default template rules apply, e.g.

```
<xsl:template match='*|/' >
  <xsl:apply-templates select='node()' />
</xsl:template>
<xsl:template match='text()|@*' >
  <xsl:value-of select='.' />
</xsl:template>
```

Parameters and Variables

- Templates may have parameters

```
<xsl:template name='function' >
    <xsl:param name='n1' />
    <xsl:param name='n2' />
...
</xsl:template>
```

- Called as follows

```
<xsl:call-template name='function'>
  <xsl:with-param name='n1' select='expr1' />
  <xsl:with-param name='n2' select='expr2' />
</xsl:call-template>
```

- Templates may also have variables
  - Analogous syntax for declaration
  - No setting of value at invocation time
  - Can be declared everywhere
- Variables and parameters can be defined only once per template
- Are globally visible

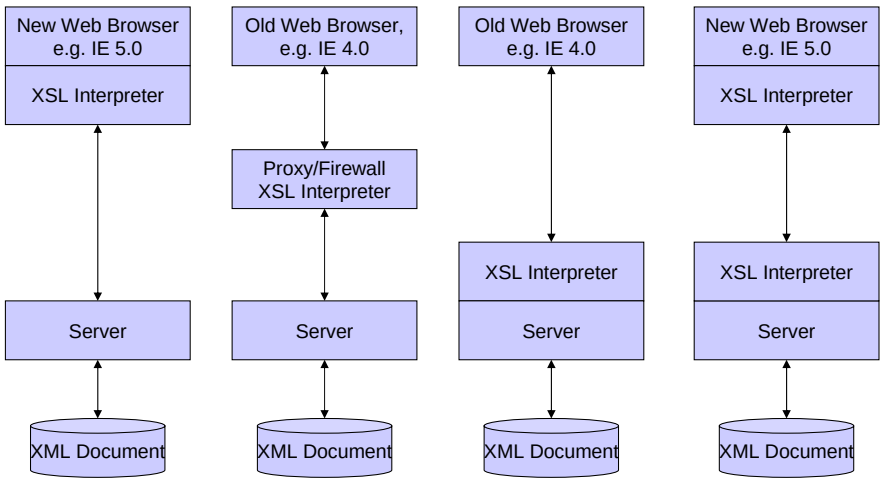
Example: Recursive Programming

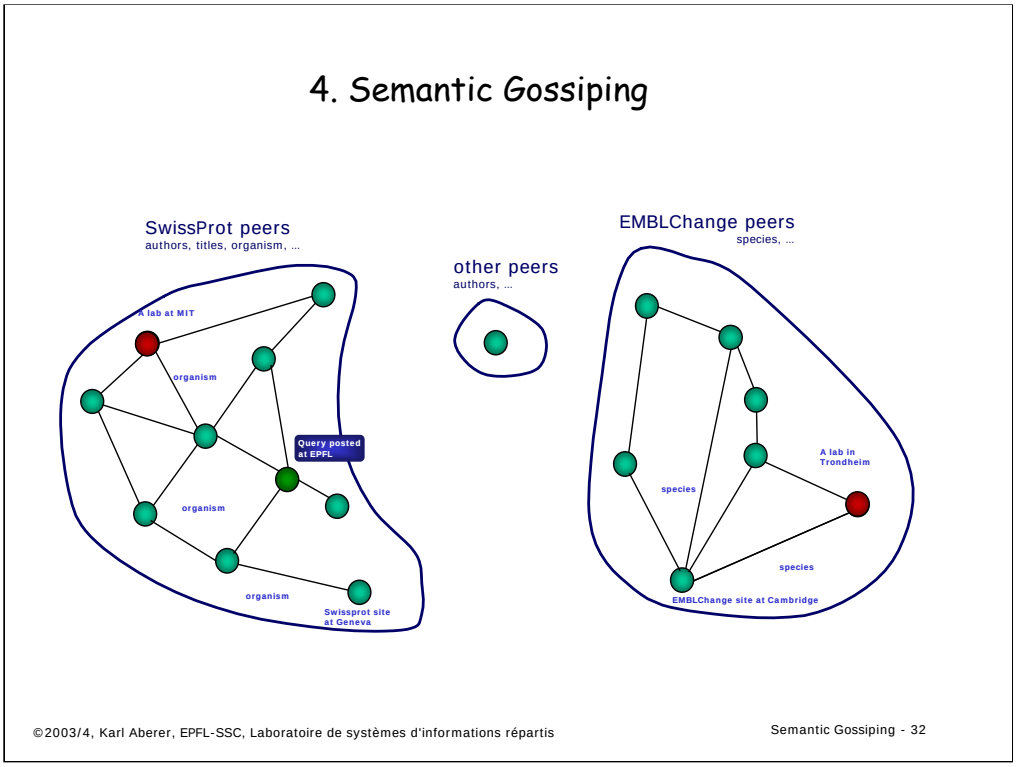
```
<?xml version='1.0' ?>
<xsl:stylesheet version='1.0'
xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>

<xsl:template match='/publications'>
<xsl:call-template name='recursive'>
  <xsl:with-param name='n' select='count(journals/paper)' />
</xsl:call-template>
</xsl:template>

<xsl:template name='recursive' >
    <xsl:param name='n' />
       <xsl:copy-of select='journals/paper[$n]/title'/>
    <xsl:if test='$n > 0'>
      <xsl:call-template name='recursive'>
        <xsl:with-param name='n' select='$n - 1' />
      </xsl:call-template>
    </xsl:if>
</xsl:template>
</xsl:stylesheet>
```

Processing Architectures for XSLT

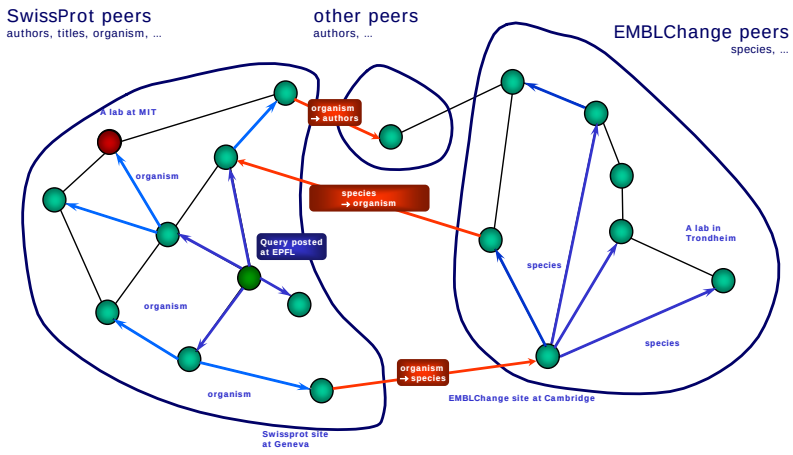




The problem we tackle: How to obtain semantic interoperability among heterogeneous data sources without relying on pre-existing, global semantic models? We believe that local agreements (i.e., translations) might be a solution.

Outline of the solution

- Local translations enabling global agreements

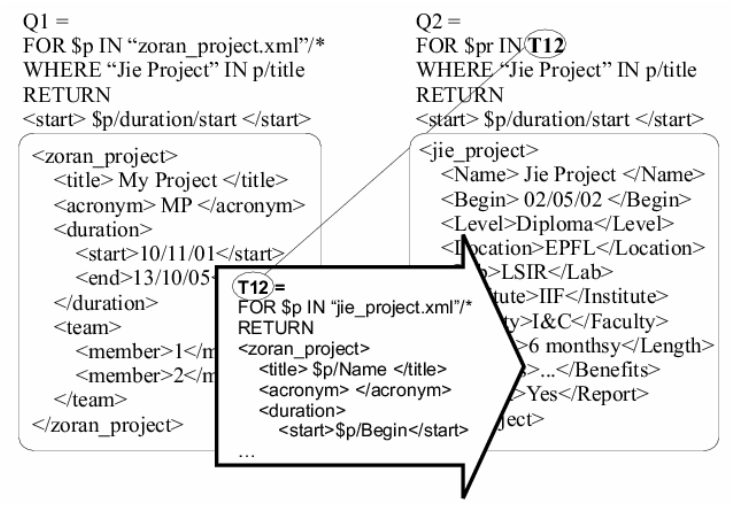


Here, we can outline several interesting properties of the resulting graph:

-In a decentralized setting, some translations might be incorrect (organism -> author above)

-Some cycles may appear (organism->species, species->organism)

On Translations



$$(T_{p_1 \to p_2}(q_{p_1}))(DB) = q_{p_1}(q_T(DB))$$

Translations between semantic domains may be written in many different ways; Here, we make use of XQuery for transforming documents. Documents from the right-hand side are transformed into documents from the left-hand side thanks to query T12. Thus, query Q1 can very easily be translated (into Q2) and be forwarded *as it* to the other domain.

Query Forwarding

- To whom shall we send the queries?
  - To peers susceptible of sending us a response…
- Simplistic solutions
  - Local Neighboring (same schema)
    - Low recall
  - Query Flooding (entire network)
    - Low precision, high network load
- Semantic Gossiping
  - Query forwarding by selecting the right peers
  - Query dependant PHBs (Per-Hop Behaviors)
  - Query / transformed queries analysis
    - Intrinsic measures (syntactic distances)
    - Extrinsic measures (semantic distances)

Starting from a web of semantic domains interconnected through local translations, the question is then: "to whom shall we send the queries"? Different approaches are possible (see above). Semantic Gossiping tries to select the peers which could be susceptible of sending back a meaningful answer.

Similarity Measures

- **Syntactic Similarity**
  - Similarity measure between an original and a transformed query.
  - Iterative computation of information loss in selections / projections.
- **Semantic Similarities**
  - Probabilistic analysis (max. likelihood) upon the correctness of translations based on feedback received

We make use of two different similarity measures to assess the quality of the translations. Then, we will only forward queries through reasonably correct translations which do not degrade the content of our query too much.

The first similarity measure is based on syntactic criteria: we simply analyze what is lost because of the translation.

The second one deals with feedback received from other semantic domains.

Semantic Similarity

- **Cycles Detection**
  - Detection of query cycles:
    - - (T1-> n) (Ai) = (Ai) √
    - - (T1-> n) (Ai) = (Aj) x
    - - (T1 -> n) (Ai) = ∅
- **Results Analysis**
  - Content-retrieval techniques:
  - classification rules to relate a returned documents to queries (extensional VS intentional expression of concepts)

The second similarity measure uses translations cycles and results received from other semantic domains in order to assess the quality of the translations (details are omitted here, please refer to research papers if interested) .

References

- **Start making sense: The Chatty Web approach for global semantic agreements,**
  Karl Aberer, Philippe Cudré-Mauroux, Manfred Hauswirth
  1st issue of Journal of Web Semantics.
- **The Chatty Web: Emergent Semantics Through Gossiping**
  Karl Aberer, Philippe Cudré-Mauroux, Manfred Hauswirth
  Proceedings of the Twelfth International World Wide Web Conference (WWW2003), 20-24 May 2003, Budapest, Hungary.
- **A Framework for Semantic Gossiping**
  Karl Aberer, Philippe Cudré-Mauroux, Manfred Hauswirth
  SIGMOD Record, 31(4), December 2002.
- **http://www.p-grid.org/**