A Logic-Based Approach to XML Data Integration

Habilitationsschrift

vorgelegt zur
Erlangung der Lehrbefugnis
für Informatik
an der
Fakultät für Angewandte Wissenschaften
der
Albert-Ludwigs-Universität zu Freiburg i.Br.

von

Wolfgang May

Freiburg, im April 2001

Submitted April 30, 2001

Dean: Prof. Dr. Gerald Urban
Referees: Prof. Dr. Georg Lausen, Albert-Ludwigs-Universität Freiburg
Prof. Dr. Rainer Manthey, Friedrich-Wilhelms-Universität Bonn

Habilitation/Venia Legendi in Computer Science: December 6, 2001

Versicherung nach § 8 (1) Nr. 5–9 HBO

Hiermit versichere ich, dass ich die vorgelegte wissenschaftliche Arbeit selbständig und ohne andere als die darin angegebenen Hilfsmittel angefertigt habe, sowie dass die Liste der wissenschaftlichen Veröffentlichungen und Lehrveranstaltungen vollständig ist.

Ebenfalls versichere ich, dass kein weiteres Habilitationsverfahren läuft, sowie dass kein Habilitationsverfahren abgelehnt wurde. Weiter versichere ich, dass keine straf- oder disziplinarrechtlichen Verurteilungen vorliegen und keine solchen Verfahren anhängig sind.

Ich versichere, dass ich von der Habilitationsordnung der *Fakultät für angewandte Wissenschaften der Universität Freiburg* Kenntnis genommen habe.

Freiburg, 30. April 2001

Abstract

In this work, a logic-based framework for XML data integration is proposed. XPath-Logic extends the XPath language with variable bindings and embeds it into first-order logic, interpreted over an *edge-labeled graph-based* data model. XPathLog is then the Horn fragment of XPath-Logic, providing a Datalog-style, rule-based language for manipulating and integrating XML data. In contrast to other approaches, the XPath syntax and semantics is also used for a declarative specification how the database should be *updated*: when used in rule heads, XPath filters are interpreted as specifications of elements and properties which should be added to the database.

Due to the close relationship with XPath, the semantics of rules is easy to grasp. In addition to the logic-based semantics of XPath-Logic, we give an algebraic semantics for evaluating XPathLog queries based on answer-sets. The formal semantics is defined wrt. a graph-based model which covers the XML data model, tailored to the requirements of XML data integration. It is not based on the notion of XML trees, but represents an XML-style (i.e., based on elements and attributes) *database* which *simultaneously* represents *individual, overlapping XML trees* as *views of the database*.

The "pure" XPathLog data model is extended with expressive modeling concepts such as a class hierarchy, nonmonotonic inheritance, and a lightweight signature concept. Information integration in this approach is based on *linking* elements from the sources into one or more result trees, *creating* elements, *fusing* elements, and defining access paths by *synonyms*. By these operations, the separate source trees are developed into a multiply linked *graph database* in which one or more *result tree views* can be distinguished by projections. The combination of data and metadata reasoning is supported by seamlessly adding XML Schema trees and even ontology descriptions to the internal database.

XPathLog has been implemented in LoPiX. The practicability of the approach is demonstrated by a case study which also serves as a running example. The first part of the essay is dedicated to an overview of the development of XML-related concepts which also motivates the design decisions of the XPathLog framework.

Contents

1 INTRODUCTION

XML has been designed and accepted as *the* framework for semi-structured data where it plays the same role as the relational model for classical databases. Specialized languages are available for XML querying, around the basic [XPa99] addressing language, e.g., XQL [Rob99], XML-QL [DFF+99b], Quilt/XQuery [CRF00, XQu01] and for transformations of XML sources, e.g., XSLT [XSL99] (also XML-QL and Quilt/XQuery can be used for generating new XML documents from given ones since their output format is XML), but at the time this paper is written, yet none of them can be seen as an XML data *manipulation* language. A proposal for updating XML will be published in [TIHW01]. For writing applications for creating and manipulating XML data, the dominating language is Java, regarding the DOM model as a data structure where applications are built on.

XML provides a uniform formalism (and by its ASCII representation also a uniform format) for electronic data interchange over the internet. Many XML applications "live" in a "predefined" setting where all participants use the same "language" and are aware of each other, e.g., in B2B (business-to-business) networks, banking, and health care. Additionally, there is a growing rate of autonomous data interchange: Information suppliers provide XML data on the Web which is then used by information brokers (e.g., search engines or market places) or users which are not previously known by the supplier. Here, the data source is autonomous in the sense that it may change the data and also the data format without notification of the users. Having numerous accessible information sources on the Web, "everybody" may extract the information which he considers to be relevant and restructure and integrate it according to his personal needs, creating a personalized view on the data sources. Here, a powerful and preferably declarative language over a flexible data model is needed.

Depending on the application, XML data can be regarded as documents (e.g., in document repositories in the publishing field), or as (excerpts from) databases (e.g., in catalogs in the B2B field, or in banking applications). The focus of the present work is on the database point of view.

The paper follows a logic-based approach: XML instances are mapped to a semantical structure which serves for interpreting *XPath-Logic* formulas. XPath-Logic is based on (i) first-order logic, and (ii) XPath reference expressions extended with variables. XPath-Logic formulas can e.g. be used for specifying constraints and reasoning on XML documents.

The Horn fragment of XPath-Logic, called *XPathLog*, provides a declarative, Datalog-style language for manipulation and integration of XML documents. The syntax and querying semantics is based on XPath. Whereas XSLT, XML-QL, and Quilt/XQuery use XML patterns for generating output (with the consequence that their output can only *generate* XML, but it cannot be used for *manipulating* an existing XML instance), our language deviates from these approaches: XPathLog works on an abstract model which represents an *XML database* supporting multiple overlapping XML trees. An extended XPath syntax is used for querying (rule bodies) *and* generating/manipulating the data (rule heads). The semantics is given as sets of variable bindings; only when final output is produced, the well-known XML syntax/model is used.

The "pure" XPathLog language also applies as a querying and data manipulation language to the "classical" XML tree model. But, for the application to information integration, the *edge-labeled navigation graph* data model shows its strengths by supporting operations such as *element fusion* for combining properties of elements which are regarded to be "the same" from different sources, *linking* which allows for (re)structuring XML trees by reusing already existing nodes, and *synonyms* which allow for reuse of access paths. Extensions to the data model combine the XML model with modeling concepts known from the object-oriented model and earlier approaches to semi-structured data such as a class hierarchy and metadata information by signatures. The latter also serve for defining *result views* of the internal *XML database*.

Structure of the paper. The essay consists of four parts:

- When carrying out such a project – the design of a programming language for a still evolving and extending data model – a detailed analysis of current and previous approaches is essential. By comparing evolving, consecutive languages and modeling concepts, important and problematic aspects and details can be identified. On the other hand, when designing, implementing, testing, and applying the language in practice, the motivation how these details have been solved, and their practical consequences become more clear.

  Thus, the first part of the work does not only sketch the basic notions, but is dedicated to an overview of XML-related concepts which are related to the present approach. The concepts are described, several details which come up repeatedly are compared, and the evolution up to now is described, also motivating design decisions for XPathLog. The basic XML concepts, i.e., XML itself and DTDs, are introduced in Section 2. Section 3 describes the development of concepts and languages around the "basic" XML documents: *XPath* provides the basic formalism for querying XML, serving as a base for the development of XML querying languages which are also described in this section. Additionally, formalisms for describing XML metadata and linking XML documents are discussed. Section 4 analyzes the different requirements when regarding XML from the document and the database point of view.

- The central concepts of the work, XPath-Logic and XPathLog are motivated and described in the second part, relating them to existing approaches:

  Section 5 defines X-structures as semantical structures which represent XML documents and presents XPath-Logic as a logic which is interpreted over X-structures. The Horn fragment of XPath-Logic, XPathLog is investigated as a rule-based XML data manipulation language in Section 6, starting with the querying semantics. Section 7 defines the semantics of XPathLog rules, focusing on the semantics of rule heads for generating and modifying XML data, and the semantics and evaluation of XPathLog programs. Section 8 compares the pure XPathLog language with other XML querying languages and related notions.

- Building upon the basic XPathLog concept, especially its application for data integration is investigated. At the same time, extending concepts are also related to the XML mainstream. Section 9 extends the XML data model by *classes*, a *class hierarchy*, and *inheritance* as known from object-oriented data models. Section 10 introduces a lightweight notion of *signatures* to describe metadata in XPath-Logic. The handling of multiple XML trees for data integration (including metadata and ontologies) is described on an abstract level in Section 11. The practicability of the approach is demonstrated by the case study describing the integration of the MONDIAL database in Section 12. Section 13 investigates the handling of XLinks.

- The forth part focusses on the implementation of XPathLog and its relationship with related approaches. Section 14 compares XPathLog with F-Logic [KLW95], which strongly influenced the design of the language (XPathLog can be seen as a crossbreed between XPath and F-Logic). Section 15 describes the LoPiX system which extends the pure XPathLog language with a programming environment providing Web access and additional built-in functionality.

A general discussion of related work and the conclusion can be found in Section 16.

The Appendix contains some DTDs and XML Schema specifications of the database. Appendix E contains the lists of definitions, examples etc. with their pagenumbers.

Publications from this thesis. Some parts of this work have already been accepted for publication. [May01c] describes XPathLog as an XML Data Manipulation Language; [May01b] focusses on the applications for data integration.

The Mondial Database

The XML MONDIAL database is used as a running example throughout this work: the "final" version, mondial-2.0 (see Appendix and [Mon]) serves for experiments with a nontrivial XML instance. The XML representations of the sources (see at [May01a]) have been used for data integration in XPathLog (cf. Section 11).

The original MONDIAL case study [Mon] has been carried out for demonstrating the use of the FLORID system [FLO98, LHL+98] as an integrated tool for wrapping and mediation [May99a], resulting in the MONDIAL database in F-Logic, Oracle, Datalog, and XML format. From this case study the wrapped sources have been exported in XML format (see Appendix D) and are now used for the XML integration case study.

MONDIAL has been compiled from the following sources:

The CIA World Factbook: The CIA World Factbook (`www.odci.gov/cia/publications/pubs.html`) provides political, economic, and social and some geographical information about the countries.

A separate part of the CIA World Factbook provides information about political and economical organizations.

Global Statistics: Cities and Provinces: The *Global Statistics* Data (`www.stats.demon.nl`) provides information about administrative divisions (area and population) and main cities (population).

Qiblih: Geographical Coordinates: The Qiblih pages (`http://www.bcca.org/misc/qiblih/latlong.html`) provide the geographical coordinates of many cities around the world.

The Terra Database: The Karlsruhe TERRA database can be seen as the origin and the "kernel" of the idea of the MONDIAL database. There, geographical and political information about countries, administrative divisions, cities, organizations, waters, mountains, deserts etc. is stored in a relational schema used for a practical training in ORACLE at Karlsruhe University.

Country Names and Codes: An additional Web page has been used which gives the country names in different languages and the country codes. TERRA uses german names and all other sources use english and local names.

2 XML: AN OVERVIEW OF BASIC NOTIONS

XML (Extensible Markup Language) has been defined as a *semistructured* data model, both suitable for the document-oriented point of view and for the database point of view. An important aspect is also that the document aspect and the data aspect may be combined in a single instance of semistructured data (thus, the terms *XML document* and *XML instance* will be used synonymously, the term *XML database* is used even more general, denoting several possibly overlapping XML instances).

Originally, the ideas of semistructured data have mainly been present in the document community, where the SGML language has been developed.

Models and languages for semistructured data have attracted a lot of interest in the database community since the mid 90s [Abi97, AQM$^+$97b, BDHS96, Suc97, KIF98]. One motivation for studying semistructured data was the immense growth and impact of the *World Wide Web*. Moreover, there was a growing need for integration of data from heterogeneous sources (e.g., legacy systems or data available from the Web), for which semistructured data provides a common data model. Typical features attributed to semistructured data include the following: the structure is irregular, partial, unknown, or implicit in the data, and typing is not strict but only indicative [Abi97]. Since the distinction between schema and data is often blurred, semistructured data is sometimes called "*self-describing*" [Bun97].

Starting from the document point of view, a document consists of its *contents*, i.e., the text, and *markup*, i.e., the structuring and annotations of the text (cf. LaTeX or RTF). Here, again, *logical* and *optical* markup are distinguished: a LaTeX *source* necessarily contains the contents together with logical markup (e.g., `\begin{itemize}\item text\item more text\end{itemize}`). Sometimes, the source also contains minor optical markup by indentation (to make the source readable, is does not influence the resulting document). Processors (e.g., LaTeX) or *stylesheets* are used to translate the *logical* markup into *optical* markup, i.e., boldfacing, vertical whitespaces, indentation, items etc., leaving the *contents* unchanged. Other languages are used for restructuring and merging documents, including their contents.

In the text processing community, the (meta)language *SGML (Standard Generalized Markup Language)* has been developed at IBM starting in 1979, and became a standard (ISO 8879:1986) in 1986. SGML defines a tree model for *documents* providing sophisticated concepts for logical and optical document markup. Additionally to the internal representations of SGML tools, there is an ASCII representation using an extensive grammar (lots of round ("(", ")") and angle ("<",">") brackets). SGML tools are widely used in publishing (for "smaller" tasks, academic researchers prefer LaTeX, which is already much easier to learn than SGML). For exchanging SGML documents, the ASCII representation is used. For handling SGML documents, languages like CSS and DSSSL have been developed.

With the *World Wide Web* as a new medium for presenting *hypertext* documents (i.e., documents which potentially associate a functionality with parts of the contents, e.g., having clickable links to other documents or starting applications), new, specialized markup requirements came up. On the other hand, the functionality needed for Web representations requires only a small subset of what has been provided by SGML. Third, the Web presentation language was expected to be much smaller, easier to parse (to be parsed by graphical Web browsers) and easier to learn than SGML (to be used by casual users). The design of *HTML (Hypertext Markup Language)*, a much stripped version of SGML started in 1989 at CERN, and HTML became a standard in 1991.

HTML specifies a set of *tags* with a predefined semantics, how to present Web pages in graphical Web browsers, such as Mosaic [NCS93] (August 1993), Netscape (April 1994), and Microsoft Explorer (May 1996).

HTML. In many aspects, HTML gives a good intuition for understanding XML: The ASCII representation of XML data and HTML share the same syntactical notions (part of SGML). Formally, HTML is a specialized SGML application, designed as a language for describing Web pages. Its grammar is specified by a *DTD (Document Type Description)*, the SGML notion of a "schema specification" (see also Section 2.2).

A HTML page is a hierarchical structure of nested *elements*, consisting of a *start tag*, e.g., <TABLE>, some contents (i.e., text and nested elements), and an *end tag*, e.g., </TABLE>. Elements may be further specified by *attributes*, e.g., <TD colspan = "2"> ... </TD>.

The HTML standard defines the *semantics* of these tags, that means, a specification how they are represented in a browser, together with a grammar which specifies how the tags may be nested and which attributes are allowed. With several extensions, HTML now provides a lot of tags for combined optical and logical markup (headers, lists, tables, hyperlinks, linebreaks, colors, fonts, pictures etc.).

2.1 XML

The development of *XML (Extensible Markup Language)* started in summer 1996 by the *XML Working Group* (a group of SGML experts lead by John Bosak (Sun Microsystems)) which worked in cooperation with the *W3C (World Wide Web Consortium)*. The design of XML was driven by the idea to have a *generic* SGML-based language for representing semistructured data in a self-describing way, i.e., the instance contains both the contents, and a description of the semantics of the contents. The latter is achieved by structuring the contents using *semantical tags*: Every XML application defines its own semantical tags which are described in a *DTD (Document Type Description)*. Then, *elements* represent objects of an application area, containing information both in the *attributes* and in the *element contents*. After some consecutive Working Drafts, the XML 1.0 Recommendation has been published in February 1998 [XML98].

2.1.1 XML Data Model

First, note that the (abstract) XML *data model* is not concerned with the "common" *representation* of XML data by ASCII files (syntactically very similar to HTML). Even XML *documents* are not necessarily given in the ASCII representation. Nevertheless, the notion of a DTD specifying attributes and elements also applies to the *abstract* XML data model in general.

The *abstract* data model of an *XML instance* (for a detailed, formal description of the XML Query Data Model, see Section 3.9.1) is a *tree*, consisting of *nodes*. The XML data model distinguishes different types of nodes, amongst them *document nodes* (the entry points of the trees), *element nodes* (the inner tree nodes), *text nodes* (leaves), and *attribute nodes* (another type of leaves). The DOM (Document Object Model) API (see Section 2.1.3) is a specification of an abstract datatype which implements this data model.

Every XML instance is associated with a unique *document node* which serves for "accessing" the XML instance (in XPath, the expression document(*url*) gives access to this node). The document node (containing some metainformation about the document) again has a unique child (which is an *element node*) which is the *root node* of the document.

- *Element nodes* (including the root node) have a name (often referred to as their "tag", e.g., *mondial*, *country*, or *city*). The *element contents* is an (ordered!) list of children, i.e., *element nodes* and *text nodes*. Additionally, elements may have a (unordered!) set of *attribute nodes* associated with them.

The XML tree structure is recursive: every element node together with its children and attributes is again a (sub)tree.

- *Text nodes* contain only text contents and have no children and no attributes.
- *Attribute nodes* have a name (e.g., *area* or *capital*), a type, and a value. The value can be either atomic, or a set of atomic values. The basic XML model distinguishes between the following attribute types:
  - `CDATA`: scalar, (nearly) arbitrary text contents[1],
  - `NMTOKEN`: scalar, token values (i.e., text without whitespaces),
  - `NMTOKENS`: multivalued, a list of tokens (separated by whitespace), e.g.,

    <country name="Switzerland" industry="machinery chemicals watches">
  - `ID`: a distinguished scalar `NMTOKEN` attribute. Its value must uniquely identify an element throughout the whole XML instance.
  - `IDREF`: a scalar *reference attribute*, its (`NMTOKEN`) value must occur as the value of an `ID` attribute somewhere in the XML instance.
  - `IDREFS`: a multivalued (`NMTOKENS`-valued) *reference attribute*, each of its values must occur as the value of an `ID` attribute somewhere in the XML instance.

    Concerning `IDREF(S)` attributes, the abstract data model does not specify how the value is actually stored – it is only required that the referencing semantics can be implemented by an application which is aware that the attribute is a reference attribute.

  The types of attributes are not given in the XML instance, but in the *document type description (DTD)* (see below).
- Additionally, there are *comment* nodes and *processing instructions* which are not considered in this work.

In general, every XML instance is of a certain *document type*, i.e., built from element nodes and attributes according to a certain specification: the corresponding *DTD (Document Type Description)* (see Section 2.2) specifies the structure of the XML instance by describing the allowed element types and attributes. An XML instance is *valid* it it conforms with the constraints specified in the DTD (which is mentioned in the unique *document node* of the XML instance).

The *abstract* XML data model described above has numerous representations:

- The DOM representation (see Section 2.1.3) which is the internal data structure of most lightweight XML tools,
- database-like representations by (proprietary) data structures in commercial systems (e.g., Tamino [Sof], eXcelon [eXc]) which are often based on the DOM specification and augmented with index structures,
- mappings to the relational model (XML extensions to relational database systems, such as Oracle's OraXML),
- implementations in *LDAP (Lightweight Directory Access Protocol)* [Ope],
- ... and the "common" ASCII representation which serves for the following purposes:
  - it is human-readable by presenting the tree as a tagged document similar to HTML, and
  - for electronic data interchange, a "standard" representation is required which can be processed by all tools (exchanging an internal data structure such as a DOM is not suitable) and communicated by the basic internet protocols.

[1] for the details of allowed characters, see the XML Standard [XML98].

DOM-based systems are often called *native* XML systems; but for the *end user*, it is not relevant which internal representation is used.

Example 2.1 (ASCII Representation of XML Instances)
Using the ASCII representation, an element node has, e.g., the form

```
<country car_code="D" capital="city-berlin">
    <name>Germany</name>
    <city id="city-berlin" name="Berlin"> ... </city>
    <city id="city-hamburg" name="Hamburg"> ... </city>
</country>
```

The country *element has a* car_code *attribute (to be declared of type* `ID` *in the DTD) which identifies it amongst* all *nodes in the document (not only amongst the* country *elements) and an* `IDREF` *attribute* capital *which is a reference to a node whose* `ID`*-declared attribute has the value "city-berlin". Its element contents is a* name *element with text contents "Germany", followed by two* city *elements. Both have an* id *attribute (declared as* `ID` *in the DTD), one of them has the value "city-berlin" which is required by the above* capital *reference.*

Definition 2.1 (Document Order)
Every XML tree defines an enumeration of its *elements*, called *document order* ($<_{doc}$) which results from traversing the tree recursively by depth-first search, listing the root element before traversing the subtree.

The notion of *document order* is extended to attribute nodes by $attr_1 <_{doc} attr_2$ if $elem_1 <_{doc} elem_2$ holds where $elem_i$ is the element to which $attr_i$ belongs. The order of different attribute nodes of one element is arbitrary. □

A formal data model for querying XML is described in Section 3.9.1. Note that there is no direct representation for use with Web browsers (except loading the plain ASCII file). For accessing XML with Web browsers, see Section 2.1.4.

2.1.2 XML Documents in ASCII Representation

One of the requirements when designing XML was that there must be a "human-readable", self-describing representation (i.e., in ASCII). This representation is derived from the SGML ASCII representation by the common tag syntax known from HTML. Since this representation is also used as the format for *electronic data interchange* over the internet, it is often regarded as *the* representation of XML data: XML applications (including XML-aware Web browsers and database interfaces) are expected to parse XML documents in this representation and to be able to export XML data in this representation.

Formally, an XML instance consists of a *prolog* (which contains some metadata about the document, in the abstract model associated with the document node) and the *root element* which in turn contains nested elements.

The *prolog* contains a *document type declaration* (not to be confused with the DTD (Document Type Description)) of the form

<!DOCTYPE *name* (SYSTEM|PUBLIC) *url*> or <!DOCTYPE *name* [*dtd*] >

which tells the *name* of the document's type (i.e., mondial) which is described by a *DTD* (see Section 2.2). The DTD can either be given *inline* by *dtd* in the above document type declaration, or by an *url* (where the DTD can be found) as an external DTD. Here, `PUBLIC` stands for a public, standard DTD whereas `SYSTEM` stands for a local, private DTD. In either case, the DTD is required to define an element type whose *name* is the *name* declared as the document type. Also, the root element of the XML instance must be an element of type *name*.

Example 2.2 (XML Instance with DTD)
The running example in this work consists of two files (for the original files, see [Mon]): The document at the url file:mondial-2.0.dtd is a DTD which declares an element type mondial (and some other things):

```
<!ELEMENT mondial (...)>
```

Then, an XML instance of the document type mondial has the form

```
<?xml version="1.0" ...>
<!DOCTYPE mondial SYSTEM "mondial-2.0.dtd">
<mondial attributes>
    elements
</mondial>
```

In contrast to HTML which requires the browsers to be very fault-tolerant with sloppy sources, the XML requirements on *well-formed* documents are much stronger:

- pairs of opening and closing tags must be correctly nested, e.g.

  ```
  <a attributes> ... <b attributes>... </b> ... </a>
  ```

- empty elements are allowed with a special syntax:

  ```
  <a attributes/>
  ```

Due to this restrictive specification, XML parsers can be implemented quickly, and *efficient* since no error-correction is required. Note that well-formedness is only concerned with the ASCII representation.

Example 2.3 (XML)
Consider the following excerpt of the Mondial *database [Mon] for illustrations (the complete DTD can also be found in Appendix A).*

```
<!ELEMENT mondial (country+, organization+, ...)>
<!ELEMENT country (name, population, city+, ...)>
   <!ATTLIST country   car_code ID #REQUIRED        memberships IDREFS #IMPLIED
                       capital IDREFS #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT city (name, population*)>      <!ATTLIST city ...>
<!ELEMENT population (#PCDATA)>      <!ATTLIST population year CDATA #IMPLIED>
<!ELEMENT organization (name, abbrev, established?, members*)>
   <!ATTLIST organization   id ID #REQUIRED   seat IDREF #IMPLIED>
<!ELEMENT abbrev (#PCDATA)>
<!ELEMENT members EMPTY>
   <!ATTLIST members   type CDATA #REQUIRED   country IDREFS #REQUIRED>

<country car_code="B" capital="cty-brussels" memberships="org-eu org-nato ...">
   <name>Belgium</name>
   <population>10170241</population>
   <city id="cty-Brussels" country="B">
      <name>Brussels</name>
      <population year="95">951580</population>
         ⋮
   </city>
```

```
        ⋮
</country>

<organization id="org-eu" seat="cty-Brussels">
  <name>European Union</name>  <abbrev>EU</abbrev>
  <members type="member" country="GR F E A D I B L NL DK SF S IRL P GB"/>
  <members type="membership applicant" country="AL CZ H SK LV LT PL BG RO EW M CY"/>
</organization>
```

A representation of this excerpt as a graph can be found in Figure 5.1.

2.1.3 The DOM API

The DOM (Document Object Model) [DOM98] is an application programming interface (API) for XML instances. It defines an abstract datatype which implements the abstract XML tree model for storing and managing XML instances. An important objective for the Document Object Model is to provide a standard programming interface that can be used in a wide variety of environments and applications. The DOM is designed to be used with any programming language. In order to provide a precise, language-independent specification of the DOM interface, the specification is defined using the *Interface Definition Language (IDL)* defined by the *Object Management Group (OMG)* in the CORBA specification. With the Document Object Model, programmers can build documents, navigate through their structure, and add, modify, or delete elements and content. Anything found in an HTML or XML document can be accessed, changed, deleted, or added using a DOM implementation (which are available e.g., for C++ and Java).

The original DOM Level 1 Specification (1998-2000) has successively been extended by DOM Level 2 (Nov. 2000) and DOM Level 3 (Jan. 2001). This reflects the ongoing work on the data model, as, e.g., described in the XML Query Data Model and Algebra (described in Section 3).

2.1.4 XML and Web Browsers

Although XML documents in their ASCII representation are very similar to HTML files, they are useless for browsers (except the XML file uses only the HTML tags, i.e., it is in fact an HTML document): for the browser, the tags have no semantics – thus, the browser cannot interpret them. It knows how to map a

```
<UL> <LI> ... </LI> ... </UL>
```

sequence to the browser window, but what to do with a

```
<country> <city> ... </city> ... </country>
```

sequence?[2]

Here, *stylesheets* can be used: For a given document type (defining a set of element types and attributes), a stylesheet defines a transformation how to map element types to XML structures according to some other DTD. If the target DTD is the HTML DTD, the result can be interpreted by a browser.

Stylesheets can be written either in CSS (Cascading Style Sheets, a simple language designed for SGML and frequently used for HTML), or DSSSL (a very expressive, complex functional language which is used for transforming SGML documents in publishing environments), or XSLT (XSL Transformations, see Section 3.4). The latter is a powerful functional-style pattern-based transformation language especially for XML, in XML syntax. XML documents on the Web may specify an XSL stylesheet (via its url) which should be applied when the document is accessed by a browser. If the browser is XSL-enabled, it will process the stylesheet and the XML instance and display the resulting HTML page.

[2]Thus, clicking on a link to an XML or DTD file at [Mon] results in an empty browser window. *View Page Source* then shows the XML ASCII representation.

2.2 DTD (Document Type Definition)

SGML and XML allow for the definition of tags. Each SGML application defines its own tags and attribute names. These are described in a *DTD (Document Type Description)*. Thus, HTML is *one* instance of an SGML language, called a *document type* – every complete HTML document refers to the DTD `"HTML.dtd"` in its prolog. The DTD describes a document type by specifying which tags are allowed, their attributes, and the allowed nestings. Roughly, the DTD corresponds to the schema definition in relational or object-oriented databases.

For an XML instance, one can either define an own DTD (as it is done for the MONDIAL database), or refer to an existing DTD. For electronic data interchange, it is necessary that all partners agree on a common DTD, e.g., car manufacturers and suppliers, medicine, or financial services use standardized DTDs.

A DTD for a document type *doctype* consists of a grammar which describes the class of documents,

- which *elements* are allowed in a document of the type *doctype*,
- which *subelements* are allowed for these elements (element types, order, cardinality),
- which *attributes* are allowed (attribute name, type, and cardinality),
- additionally, *entities* may be defined. Since entities are not a modeling concept, but merely act as macros for writing a document, they are not considered in the sequel.

The *structure* of the *contents* of *elements* is defined by *element type declarations*

<!ELEMENT *elem_type contentsmodel*>

where

- if *contentsmodel* = `EMPTY`, elements of type *elem_type* are empty, i.e., do not have any contents (but may have attributes).
  Note that an *empty* element, e.g., <foo attr="..."/>, is different from an element with *empty contents*, e.g., <foo attr="..."/> </foo>.
- if *contentsmodel* = `(#PCDATA)`, elements of type *elem_type* have only *text contents* and perhaps attributes,
- if *contentsmodel* = (*expression*) where *expression* is a regular expression over element names, the structure of a complex element type is described. The following constructs are used:
  - ",": sequence,
  - "|": exclusive-or (choice),
  - "*": arbitrary often,
  - "+": arbitrary often, at least once,
  - "?": optional (0 or 1 times).
- if *contentsmodel* = (`#PCDATA`|$elem\_type_1$|...|$elem\_type_n$)*, an element with *mixed contents* is described, i.e., elements of this type may contain text children and children of the given element types in *arbitrary ordering and nesting*. It is not possible to give a more detailed specification of element types with mixed contents in a DTD.
- if *contentsmodel* = (`ANY`), elements of this type may have arbitrary contents (even, element types which are not named in the DTD).

The allowed *attributes* of elements are declared by *attribute declarations*. An attribute declaration is of the form

```
<!ATTLIST elem_type
          attr_name1 attr_type1 attr_constr1
             ⋮          ⋮           ⋮
          attr_namen attr_typen attr_constrn>
```

where for an element type, all its attributes are described by their name, their attribute type and a constraint on their cardinality. DTDs allow only for a very restricted set of attribute types which has already been mentioned for the XML data model:

- `CDATA`,
- `NMTOKEN`,
- `NMTOKENS`,
- `ID`,
- `IDREF`,
- `IDREFS`,
- additionally enumerations: if

  $$\mathit{attr\_type} = (\mathit{const}_1|\ldots|\mathit{const}_k)\ ,$$

  the attribute is scalar, with token type; additionally, it is required that the value is one of the enumerated values, e.g.,

  ```
  <river name="Rhein">
     <to type="sea" water="sea-north-sea"/></river>
  ```

  with the DTD fragment

  ```
  <!ATTLIST to
            type   (river|lake|sea)  #REQUIRED
            water  IDREF             #REQUIRED >
  ```

  or a string-valued enumerated attribute, e.g., for the state codes of a US database:

  ```
  <!ELEMENT city (...)>
  <!ATTLIST city state (AL|AK|AR|AZ|CA|...) #REQUIRED
                 ...      >
  ```

Whereas the *attribute type* specifies the type *and* in some sense the maximal cardinality of attributes, the *attribute constraint* $\mathit{attr\_constr}_i$ primarily specifies the minimal cardinality. Additionally, it is used for defining default or mandatory values:

- `#REQUIRED`: the attribute must be given for each instance of the element type.
- `#IMPLIED`: the attribute is optional.
- `#FIXED` *value* (where *value* is a value allowed for $\mathit{attr\_type}_i$): the attribute has this value for *all* instances of the element type (monotonic value inheritance). Note that for multivalued attributes, lists are allowed.
- *default*: the attribute will implicitly be present for each instance of the element type. If a value is given in the document, it is used, otherwise the default is used (a restricted kind of non-monotonic value inheritance). Note that the inheritance is overriding, not accumulating.

Remark 2.1
`FIXED` attributes are especially used when assigning values to attributes with a meta-semantics defined in several namespaces providing special XML-related functionality, e.g., XLink (see Section 3.8). □

Note that a *cardinality > 1* still means that there is at most one attribute value for every instance of the element type, but that the attribute is of a *collection type*, e.g.

<country name="Switzerland" industry="machinery chemicals watches">

where industry is defined as `NMTOKENS`.

Note also that (i) there is no order of attributes, but (ii), for "multivalued" attributes, the values are sometimes regarded to be ordered (e.g., in [TIHW01]).

Validity. An XML instance is valid wrt. a specified DTD if it satisfies the constraints expressed in the DTD (if the document is in its ASCII representation, *well-formedness* is basically required). Note that this also requires that all values of type `IDREF` must match the value of some `ID` attribute.

Example 2.4 (DTD, Fixed Values)
Consider the following fragment of a DTD:

<!ELEMENT *name* EMPTY>
<!ATTLIST *name* $attr_1$ CDATA "*value*"
$attr_2$ IDREFS #FIXED "id_1 id_2">

*By having a **FIXED** attribute value for a reference attribute, this requires every document of this document type which contains an instance of the element type* name *to contain at least two elements with the **IDs*** id_1 *and* id_2.

Discussion. DTDs are a heritage from the SGML area, tailored to describe the document structure. Here, the notion of element *types* is present for specifying the structure of documents, but not as a *modeling concept*. Some typical schema issues from the database area are not covered:

- Datatypes: the only literal types are `CDATA/PCDATA` and `NMTOKEN(S)`.
- Cardinalities: instead of min/max cardinalities as, e.g., in the ER model or in UML, only optional/required and scalar/multiple (by iteration) can be specified.

A significant weakness is that it is clumsy to specify that an element must contain some subelements once in *arbitrary* order:

Example 2.5 (DTD: Non-ordered Subelements)
A country *element must have one* area *subelement, one* population *subelement and one* gdp *subelement in any order:*

<!ELEMENT country ((area,population,gdp)|(area,gdp,population)|
(population,gdp,area)|(population,area,gdp)|
(gdp,area,population)|(gdp,population,area))>

Note that

<!ELEMENT country ((area|population|gdp)*)>

would allow every element to occur several times.

Additionally, the DTD does not know anything like a *class* or *type hierarchy*. It is not possible to specify one type as an extension of another (except the purely textual use of *entities* as macros in the DTD code). Thus, the inheritance concept is actually restricted. Another problem with DTDs is that they are not in XML syntax.

The W3C XML Schema working draft [XML99a] (cf. Section 3.7) provides an XML syntax database style formalism for specifying XML metadata, introducing simple datatypes, complex datatypes, a datatype extension hierarchy etc.

2.3 Namespaces

Every application defines its own element and attribute names. Especially, several XML-related concepts, e.g., XSLT (see Section 3.4), XMLSchema (see Section 3.7) and XLink (see Section 3.8) use predefined names which may occur in combination with names used by the application. Additionally, there are standardized namespaces for several application areas, e.g., medicine, banking etc. whose elements and attributes are understood by applications in the respective area.

Here, XML introduces the *namespace* concept: a node (element and attribute) does not only have a name (e.g., country), but also a namespace (e.g., mondial). Then nodes sharing the same name, originating from different sources (also having totally different structure) can be distinguished. Every namespace is associated with a url where additional information can be found.

Example 2.6 (Namespaces)
Consider an application which combines the MONDIAL *database with a linguistic database. The* mondial:language *element is defined as*

```
<!ELEMENT country (..., language*, ...)>
<!ELEMENT language (#PCDATA)>
<!ATTLIST language percentage CDATA #REQUIRED>
```

and contains information, where a language is spoken by how many people. In contrast, a linguistics:language *node*

```
<!ELEMENT language (name, derivation, period, model, syntax, grammar, ...)>
```

describes a language from the linguistics point of view:

- derivation *states from which language(s) it is derived, possibly using additional subelements which derive the language's features which have been influenced by some other language,*
- period *describes when the language has developed and how it has changed (e.g., medieval german, modern german),*
- model *describes if it is a word-based, syllable-based, or letter-based,*
- syntax *describes if it is written from left to right etc.,*
- grammar *may be a complex element containing information on grammar patterns (existence of definite articles, pronouns, cases, tempi etc).*

An application which uses both sources will then distinguish mondial:language and linguistics:language nodes; similar for mondial:name and linguistics:name.

A node which is associated with a namespace inherits the namespace information to its subelements and attributes, i.e.,

```
<mondial:country car_code="B">
<name>Belgium </name>
```

```
<language percentage="56">Dutch</language>
<language percentage="32">French</language>
<language percentage="1">German</language>
</mondial:country>
```

implicitly associates the mondial namespace also with the car_code attribute, and the name and language subelements.

Distinguishing between application-level concepts and meta-level concepts. Namespaces are important when XML documents are augmented with elements carrying a meta-level semantics: XML documents can e.g., contain formatting information, cf. *XSL formatting objects* (see Section 3.4) which describes the optical markup. Another example is the extension with *linking* functionality using the *XML Linking Language* which is described with the MONDIAL example in Sections 3.8 and 13. In these cases, the (predefined) meta-level concepts are intended to be interpreted application-independently by the used tools whereas the actual application logic is concerned with the application-level concepts.

Information integration. In this case, information from several sources has to be integrated on the application level. In general, the concepts defined in the sources are overlapping. The sources may use different *ontologies* (where potentially the same names are used for the same concept, or also for different concepts). Applications use the namespaces for distinguishing which ontology has to be used for interpretation of nodes. Information integration using namespaces is further considered in Sections 11 and 12.

3 XML: FURTHER NOTIONS

3.1 Overview

Today, nearly all languages in the XML world are based on XPath [XPa99] (see Section 3.2). The first versions of XPath have been developed as *W3C XSL Pattern Language* [XSL98][1] in the August 1998-April 1999 *W3C XSL (Extensible Stylesheet Language)* working drafts (note that the Dec. 1998 WD did not yet use the terms of *axes* and *location paths*). XSL Patterns were mainly employed in the *W3C XSLT (XSL Transformations)* part [XSL99]. In parallel, the *W3C XPointer* working drafts [XPt00] (*XML Linking* (April 1997), *W3C XPointer* (March 1998)) coined the term *location paths*. The first *W3C XPath* working draft in July 1999 [XPa99] combined the XPointer and XSL Pattern notions into *XPath* which since then serves as a unified base for *W3C XSLT* (WD July 1999, see Section 3.4), *W3C XPointer* (WD Dec. 1999, see Section 3.8), *W3C XQuery* [XQu01] (February 2001, see Section 3.11), and other concepts. A formal semantics of XSL Patterns/XPath can be found in [Wad99a, Wad99b].

XQL (XML Query Language) [RLS98, Rob99] (see Section 3.3) was an early proposal (1998, non-W3C) for a simple querying language which has been developed in parallel to the *W3C XSL Pattern Language* (still not using the terms *axis* and *location path*). The experiences with XQL influenced the design of XPath and subsequent concepts. XQL has been implemented in [HM99].

W3C XSLT (XSL Transformations) [XSL99] (see Section 3.4) has been developed as the W3C transformation extension to the XSL Pattern Language. Since the July 1999 WD (after the separation of XSL Patterns into XPath), XSLT is formally based on XPath. XSLT embeds XPath in a rule-based functional-style language for generating an XML result tree. The syntax of the functional part is XML where XPath expressions are embedded as attribute values or text contents, using the reserved namespace xsl: which provides specialized "command" elements. There are multiple implementations, both in the research and in the commercial field.

The first commercial products have been developed based on these early proposals: Tamino [Sof] and Excelon [eXc] provide XML platforms for storing, querying, transforming, and integrating XML data, combined with additional internet-related and database-related functionality. In the early development stages, both products implemented XQL and XSLT for querying and transformation. By now, they migrated to XPath/XSLT.

XML-QL [DFF+98, DFF+99b, DFF+99a] (see Section 3.5) has been presented in 1998 in a non-W3C proposal for an *XML querying and transformation* language. The design (and implementation) of XML-QL has been influenced by the STRUDEL/STRUQL [FFLS97, FFK+98] project. In contrast to the *W3C XSL Pattern Language* which defined the W3C state of the art at that time, it was designed to provide a better support for data-intensive applications such as joins and aggregates, and for constructing new XML data. The basic idea was influenced by SQL-like languages, partitioning XML-QL queries into a *selection part* and a *construction part*:

```
WHERE xml-pattern
IN url
CONSTRUCT xml-pattern
```

[1]We refer to W3C working drafts and recommendations by mentioning W3C explicitely at the first occurrence to distinguish them from non-W3C proposals (which were often submissions to W3C, intended to influence the W3C developments). Working drafts and recommendations on all W3C concepts can be found at [W3C]. Non-W3C concepts are, e.g., XQL, XML-QL, Quilt, YAXQL.

In contrast to the navigation/path-based approach of XSL Patterns/XQL, XML-QL uses XML patterns in the `WHERE` part which are matched against the queried document for extracting variable bindings which are then used in the generating clause. Although, compared to XSLT, the transformation functionality is restricted since it is not possible to operate on the result set of a previous step. XML-QL has been implemented in [DFF+99c]. Several systems use XML-QL, e.g., SilkRoute [FTS00], or MIX [BGL+99] (see also Section 16.1).

`IDREFS` attributes are a crucial problem with XQL and XML-QL: Their values are a sequence of ids, separated by whitespaces, e.g., memberships="org-EU org-UN org-NATO ...", denoting that the attribute references a set of objects by enumerating their ids. Since both XQL and XML-QL do not provide a special dereferencing operator but use joins for dereferencing, it is not possible in these languages to split `IDREFS` attributes in their individual references. The XSL Pattern Language defines the id(...) function which implicitly splits `IDREFS` attributes and follows each of the references. Nevertheless, the same problem remains for `NMTOKENS` (see also Section 3.2.2).

YAT/YATL (*Yet Another Tree Model/Language*) [CDSS99] is a pre-XML proposal, already using SGML and DTDs. Its trees provide a unified model for relational, object-oriented (ODMG) and semistructured/document data (SGML). The YATL language follows a rule-based design for complex objects in the style of MSL or F-Logic. In [CCS00], the YAT system is turned into an XML system for data integration.

In [FSW99], XQL, XML-QL, and the languages YATL [CDSS99] and *Lorel* [AQM+97a,GMW99] have been compared and essential features of an XML querying language have been identified. In Section 8.1.1, it is shown that XPathLog satisfies these requirements.

Based on the experiences with XSL Patterns/XQL, XSLT, and XML-QL, the requirements on an XML querying language have been stated in the *W3C Query Requirements* working draft [XMQ01c] (see Section 3.6). Roughly, the results concerning this work are

- the XML Query Language must be declarative, not enforcing a particular evaluation strategy, and
- it should have several language bindings. One language syntax must be convenient for humans, and *one syntax must be expressed in XML*. The idea here is – similar to XSLT – that queries are XML instances and thus can be manipulated themselves by XML languages.

Schema and metadata aspects have not been dealt with in the above languages. From the database point of view, DTDs (which are a heritage from the SGML world) are not sufficient; additionally, since their syntax is not XML, they do not fit into the picture: a metadata format in XML is favorable. This requirement is satisfied by *W3C XML Schema* [XML99a] (see also Section 3.7) which provides a database-oriented metadata language for XML in XML syntax. XML Schema defines (basic) *datatypes* and complex *structures (complexTypes)*; the latter are then used as element types.

Based on the above requirements and the experiences with XML Schema, the *W3C XML Query Data Model* [XMQ01b] (see Section 3.9.1) and *W3C XML Querying Algebra* [XMQ01a] (see Section 3.9.3) have been defined. The XML Query Data Model defines formally the information contained in the input to an XML Query processor in terms of trees. Especially, it supports the explicit handling of multiple documents for data integration. The algebra serves for giving the semantics for XML query languages using the XML Query Data Model. Being formally defined by operators, is also serves for formal investigations (e.g., query optimization) of XML query languages (similar to the relational algebra for SQL optimization). The type system behind the algebra is based on that of XML Schema.

In some sense incorporating an "evolution step", Quilt [CRF00, RCF00] (see Section 3.10) is the first query language that extends XPath syntax with higher-level constructs (as has already been done in XML Schema and XLink for other tasks). Quilt queries consist of a series of `FOR - LET - WHERE - RETURN` clauses (FLWR; pronounced "flower"):

FOR *variable* IN *xpath-expr*

LET *additional_variable* := *xpath-expr*
WHERE *filters*
RETURN *xml-expr*

The `FOR - LET - WHERE` clause forms the extraction part, generating a set of variable bindings which are used in the `RETURN` clause to construct an XML tree. FLWR expressions may be nested in many ways.

Quilt has been influenced by earlier languages, i.e., SQL/OQL for the functional design, XPath/XQL for addressing nodes, and especially XML-QL for the structure of the FLWR expressions. An important difference to XML-QL is that Quilt uses XPath expressions for generating variable bindings where XML-QL uses XML pattern matching.

By *embedding* XPath into the higher-level construct of an FLWR expression, there can be many equivalent, syntactically very different queries. Thus, the XML query algebra described above will find an application here. An extensible, open-source Quilt implementation in Java is available at [Sah00].

With some minor revisions, Quilt (using the XPath 2.0 semantics now) has become the *XQuery* Language [XQu01] (Feb. 2001 Working Draft; see Section 3.11). Current work now concerns the details in the relationships with other XML activities:

- Definition of the semantics of XQuery in terms of the XML Query Algebra (a mapping from XQuery expressions to the Algebra is given in [XQu01]),
- Currently, the type system of XQuery is the type system of XML Schema. As already remarked for XML Schema, the type systems of XML Schema, the XML Query Algebra, and XQuery have to be aligned. As a consequence, also the specification of the DOM API is expected to be adapted.
- XQuery does not yet – as required in the XML Query Requirements [XMQ01c] – provide a syntax which is completely in XML. Nevertheless, an XQuery version which embeds XPath into XML-syntax XQuery FLWR expressions (similar as XSLT-style combines XPath and its own elements into XML syntax) is just syntactic sugar.

The first (to the knowledge of the author) XML query language which *is* in XML syntax has been presented in [Moe00]: *YAXQL* defines an XSLT-style query language, using the xql: namespace (see Section 3.14).

The above-mentioned commercial products Tamino [Sof] and Excelon [eXc] were adapted from XQL to XPath as the basic query language; also extending XSLT with extension functions. System integration is supported by Java APIs providing DOM access to the database contents, and for XPath querying from Java environments. Additionally, they provide access to the metadata by specific tools.

The first "XML-native" language for *updating* XML database contents has been defined in *XUL (XML Update Language)* [eXc00] which extends XSLT with data manipulation constructs (see also Section 3.13).

A proposal for extending XML querying languages based on variable bindings will be published in [TIHW01] (see Section 3.12)[2].

As a "world-wide" data format, XML also provides a concept for expressing links between XML documents: The *XLink* part of *XLL (XML Linking Language)* (see Section 3.8) defines elements with hyperlink semantics, based on XPath/XPointer. Currently, no querying language has been defined which provides special semantics for handling XLinks. A proposal for the handling and navigation of XLinks in queries is presented in Section 13.

In the following subsections, more detailed descriptions of the above languages and concepts are given, leading to first conclusions in Section 3.16.

There are two aspects which emerge for nearly all XML-related concepts:

[2] thanks to the authors for providing me with a copy before.

- handling of `IDREFS` attributes (an `IDREFS` attribute is a *single* value of the form "$id_1\ id_2\ \ldots\ id_n$", referencing *several* nodes),
- the duality between elements with text contents and literals, which comes up in XPath and is "solved" in XML Schema, and
- the result trees may contain dangling reference attributes (cf. Section 11.8).

The background for discussing these XML-related concepts is made up by the search for a data model and a language for use in XML databases where XML data should be restructured and integrated.

3.2 XPath

XPath [XPa99] defines the basic addressing mechanism in XML documents, which is employed by most XML querying languages. The expressions which are defined by XPath are called *location paths*. Every location path declaratively selects a set of nodes from a given XML document.

The syntax follows the UNIX Directory Notation, e.g.,

/mondial/country/city/name

addresses all nodes N such that N is a *name* subelement of some *city* element which in course is a subelement of some *country* subelement of a (the unique) *mondial* subelement of the root element.

The combination of the original URL specification with XPath expressions is defined in the XPointer standard (see Section 3.8), e.g.,

file:/home/dbis/Mondial/mondial.xml#mondial/country/city/name

is an XPointer which points to *name* subelements of *city* elements in the document at the url `file:/home/dbis/Mondial/mondial.xml`.

Navigation: location steps. XPath is based on navigation through the XML tree by path expressions of the form //*step*/*step*/.../*step*. Formally, the input to every *location step* is a *node set*, called the *context* (the input to the first step ist the set containing only the document node). From this set, a new node set (called *result set*) is computed which then serves as input for the next step. For this computation, the input node set is processed, evaluating the location step for every node in it, appending its result set to the overall result, and proceeding with the next node. Every single step is of the form

axis::*nodetest*[*filter*]*.

which specifies that navigation goes along the given *axis* in the XML document. The *axis* specifies the tree relationship between the nodes selected by the location step and the current *context node*. Along the chosen axis, the *nodetest* specifies the node type and the name of the nodes to be selected. From these, the ones qualify which satisfy the given *filter* (which in turn contains predicates over XPath expressions). If more than one filter are given, they are applied iteratively.

The semantics of XPath expressions is defined in terms of *node sets*, i.e., *unordered* forests. Only *during the evaluation of individual navigation steps*, there is an temporary node list (called *context*).

Axes. For every navigation step, the axis specifies the direction of navigation in the tree. All *forward axes* (denoted by (f)) enumerate the nodes in document order (cf. Definition 2.1), whereas all backward axes (b) enumerate them in reverse document order:

Definition 3.1 (XML Axes)
Given an element, every axis defines a list of nodes:

- self axis (f): contains exactly the element itself,
- child (f): enumerates all subelements of the element,
- descendant (f): enumerates all subelements by depth-first search (enumerating the root of a subtree before traversing the subtree),
- parent (f): contains exactly the parent of an element,
- ancestor (b): enumerates the ancestors, starting with the parent,
- following-sibling (f): enumerates the following siblings of the element,
- preceding-sibling (b): enumerates the preceding siblings of the element,
- following (f): enumerates all nodes following the current element in document order,
- preceding (b): enumerates all nodes preceding the current element in document order,
- analogous, descendant-or-self (f), ancestor-or-self (b),
- attribute: enumerates all attributes of the element. Here, the enumeration order is not relevant. □

The most frequently used axes are abbreviated as

- *path*/nodetest for *path*/child::nodetest,
- *path*//nodetest for *path*/descendant-or-self/child::nodetest, and
- *path*/@nodetest for *path*/attribute::nodetest.

Remark 3.1
Note that *path*//nodetest is different from *path*/descendant::nodetest. Let n be a node addressed by *path*:

path//node()[3]   which is equivalent to   *path*//descendant-or-self/child::node()[3]

selects all third children from any node x on the *descendant-or-self* axis (the context for selecting "[3]" are the children of x) wrt. n, whereas

path/descendant::node()[3]

selects the third descendant of n. □

Nodetests. The second part of a step is the *nodetest*. It specifies the node type and the name of the nodes to be selected by the location step. Every axis has a principal node type: For the attribute axis, the principal node type is attribute, for all other axes, the principal node type is element, including `PCDATA` (text) elements. In the location steps considered in this work, the node test is one of the following:

- a name, e.g., *path*/child::city.../..., which selects all subelements of the context node having the given name (here, all *city* elements), or
- the test on text contents, i.e., *path*/text(), which selects all `PCDATA` subelements of the context node,
- the test on element contents, i.e., *path*/node(), which selects all element children of the context node,
- * (wildcard): selects all element nodes.

Filters in location steps. Using filters, the node list obtained from the nodetest is further restricted – selecting only those nodes which satisfy the given filter. When evaluating a filter, the node list selected by the above steps (navigation along an axis and applying the nodetest) is called the *context*, and the currently processed node is the *context node*. The filter contains *predicates* over *expressions*, i.e., terms of the following form:

- booleans over predicates,
- literals (strings, numbers), arithmetic expressions over numbers, string operations,
- function calls: they serve e.g., as aggregation functions, or for stating conditions on the relationship between the current *context node* and its *context*:
  - last(): returns n such that n is the size of the context,
  - position(): returns the index of the context node in the current context (i.e., "5" if the context node is the 5th node in the node list which remained after evaluating the *nodetest*). Predicates containing the last() or position() function are called *proximity position predicates* in [XPa99, Chap. 2.4]. The filter [position()=i] may be abbreviated by [i].
  - count(*nodeset*): returns the number of nodes in *nodeset* (which results from evaluating a location path wrt. the *context node*), e.g., count(city) returns the number of *city* subelements of the current context node,
  - id(*expr*): returns the node(s) in the current XML instance whose id(s) result from evaluating *expr* wrt. the *context node*), e.g., id(@capital) returns the object which is referenced by the *capital* attribute of the context node,
  - additionally, string functions, data conversion functions, and boolean functions,
- location paths: location paths evaluated as predicates evaluate to true if their result set is nonempty.

Inside filters, *relative* or *absolute* location paths may be used:

- relative location paths are evaluated wrt. the current context node, e.g.,

  path/*axis*$_1$::*nodetest*$_1$[*axis*$_2$::*nodetest*$_2$...].

- similar to the UNIX directory notation, absolute location paths begin with "/". They are evaluated wrt. the root node of the XML document, e.g.,

  path/*axis*$_1$::*nodetest*$_1$[/*axis*::*nodetest*$_2$...].

Comparison Predicates. The comparison predicates $=$, $<$, $>$ are extended to (node) sets as follows: *locationPath*$_1$=*locationPath*$_2$ evaluates to *true* if there is a node in the node list which results from evaluating *locationPath*$_1$ and a node in the node list which results from evaluating *locationPath*$_1$ such that the result of performing the comparison on the string-values of the two nodes is true (if necessary, after suitable conversions from element contents to literals, for details see [XPa99, Section 3.4]; see also Section 5.6).

Filter application. Note that filters which are applied iteratively on a result set do in general not commute if they use *proximity position predicates*. It is also not possible to combine them into a single filter:

Example 3.1 (Proximity Position Predicates)
Consider the XPath expression

```
/descendant::country[name="Germany" or name="France"]
                    /descendant::city[population > 500000][position()=3]
```

which selects the third city in Germany which has a more than 500000 inhabitants, and the third city in France which has a more than 500000 inhabitants: the computation starts with the document node. From this, the first location step

/descendant::country[name="Germany" or name="France"]

first creates an intermediate result list enumerating all descendants. Applying the nodetest *(country), all element which are of of type* country *are removed from the list. Then, the filter is applied, keeping only the list (france, germany) which is converted into a set, being the first intermediate result set.*

Then, iterating over this set (now in arbitrary order!), for each element the second location step *is applied:*

/descendant::city[population>500000][position()=3]

Here again the first intermediate result list contains all descendants. Then, the list is pruned to contain only the city subelements. Next, the first filter is applied, dropping all cities with less than 500000 inhabitants. From the list resulting from this step, the third element is taken to the second intermediate result set. So, after evaluating the location step for both countries, the result set consists of two city *elements.*
In contrast, for the expression

/descendant::country[name="Germany" or name="France"]
/descendant::city[position()=3][population>500000]

the evaluation of the second location step is completely different: Again, for both countries, the first intermediate result list contains all descendants and is then pruned to contain only the city subelements. Then, the first, the filter [position()=3] is applied, retaining only the third city. Applying the filter [population>500000], this element survives only if it has more than 500000 inhabitants. Thus, probably the result is empty.
For the combined filter,

/descendant::country[name="Germany" or name="France"]
/descendant::city[position()=3 and population>500000]

the outcome depends on the internal evaluation.

Filters over location paths. Above, filters have been applied *inside* location steps, where the "input" to the filter was the list selected by *axis::nodetest* and possibly filters. Filters can also be applied to result sets obtained from a sequence of location steps:

(//*step*/.../*step*)[*filter*]

In case that *filter* does not use *proximity position predicates*, this expression is equivalent to

//*step*/.../*step*[*filter*] ,

i.e., applying the filter in the innermost *locationStep*. If *proximity position predicates* are used, the result is in general different:

Example 3.2 (Proximity Position Predicates cont'd)
Consider again the XPath expression

/descendant::country[name="Germany" or name="France"]/descendant::city[position()=3]

which selects the 3rd city in France and the 3rd city in Germany. In contrast,

(/descendant::country[name="Germany" or name="France"]/descendant::city)[position()=3]

selects the 3rd city among all cities in Germany or France: first, *all these cites are selected in a result set, and then the third element of the set wrt. document order is selected.*

The result of (...) is a result set, thus, for evaulating proximity positions, the global document ordering is used.

Example 3.3 (XPath)
The following XPath expressions may be evaluated wrt. the MONDIAL *database:*

- *The absolute location path*

  /mondial/country//city/name

  selects all name *subelements from* city *descendants from any* country *child of any* mondial *child of the root element.*

- /mondial/country//city/name/text()

  selects the text contents of these elements.

- /mondial/country[name = "Germany"]//city/name/text(), and
  /mondial/country[name/text() = "Germany"]//city/name/text()

  do the same, but only for cities in Germany. Here, implicitly the text contents of the name *element is compared to the string "Germany".*

- //city[population > 5000000]/name/text()

  selects all names of cities which have more than 5000000 inhabitants. Similar to above, the comparison is implicitly applied to the text contents *of the* population *subelement of the* city *element. The query*

  //city[population[@year < 1990] > 5000000]/name/text()

  selects all cities which satisfied the above condition even before 1990. Here, the population *subelement is simultaneously interpreted as an integer literal (in the comparison) and as an element node (when querying its* year *attribute). This problem is investigated in detail in Section 5.6.*

- /mondial/country/@car_code

  selects all car code *attribute nodes of* country *elements.*

- /mondial/country[inflation]

  selects all country nodes (i.e., the whole subtrees) which have an inflation *subelement.*

- *The following expression uses an absolute path in a filter, selecting the city whose id is the value of* seat *attribute of the organization whose name is "EU":*

  //city[@id = /mondial/organization[name="EU"]/@seat]

  Note that

  /mondial/organization[name="EU"]/@seat

  does not *select this city element, but simply the* ***IDREF*** seat *attribute node of the element representing the* EU, *i.e., the value "city-belgium-brussels". In contrast, the location path*

  id(/mondial/organization[name="EU"]/@seat) *would also select the above* city *element.*

- *The* memberships *attribute of* countries *is defined as* ***IDREFS****, thus, it is multivalued:*

  /mondial/country[car_code="D"]/@memberships

  selects the ***IDREFS*** *attribute value, i.e., "org-EU org-NATO ...".*

  id(mondial/country[car_code="D"]/@memberships)

  selects all elements which have one of these ids, i.e., all organizations where Germany is a member.

Proximity positions:

//organization[3]/@seat

selects the value of the seat (reference) attribute of the third organization (if the third organization has no seat attribute, the result is empty).

(//organization/@seat)[3]

selects the third value which is a seat attribute of an organization (since not each organization has a seat attribute, this may be the seat attribute of the forth or fifth organization). Note that //organization/@seat[3] selects nothing, since there is no organization which has a third seat attribute.

id((//organization/@seat)[3])

selects the city which is the seat of the third organization, and

(id(//organization/@seat))[3]

selects the third city (wrt. document order) which is a seat of an organization.

3.2.1 Formal Semantics of XPath

A formal semantics of XPath has been given in [Wad99a, Wad99b]. This semantics also forms the base for the extension from XPath to XPath-Logic and XPathLog in Section 5.

The mappings $\mathcal{S}$, $\mathcal{Q}$, and $\mathcal{E}$ define the semantics of XPath patterns, predicates (filters), and expressions as shown in Figure 3.1. It uses auxiliary mappings $\mathcal{A}[[a]]$ which enumerate the axes, $\mathcal{P}(a)$ which gives the axes' principal nodetype, and $\mathcal{D}$ which gives the direction (forward/reverse) of axes.
This semantics does not completely specify all XPath constructs:

- filters may be of the form [a/b/c = "foo"] containing a "=" predicate where a/b/c must be evaluated by $\mathcal{S}$ and "foo" is a constant which is evaluated by $\mathcal{E}$.

- expressions of the form (//country//city[population>100000])[position()=2] are allowed in XPath, but for the *primaryExpr* "(...)", $\mathcal{S}$ does not define a semantics.

- the semantics of the id() function is not defined.

Nevertheless, the intention is clear; when comparing it to the semantics of XPathLog in Section 5.5, we straightforwardly extend it to the missing constructs in the "sense of [Wad99b]".

$$
\begin{array}{lcl}
\mathcal{S} & : & Axis \rightarrow Pattern \rightarrow Node \rightarrow Set(Node) \\
\mathcal{S}^a[[p_1 \,|\, p_2]]x & = & \mathcal{S}^a[[p_1]]x \cup \mathcal{S}^a[[p_2]]x \\
\mathcal{S}^a[[/p]]x & = & \mathcal{S}^a[[p]]root(x) \\
\mathcal{S}^a[[p_1/p_2]]x & = & \{x_2 \mid x_1 \in \mathcal{S}^a[[p_1]]x, x_2 \in \mathcal{S}^a[[p_2]]x_1\} \\
\mathcal{S}^a[[a_1 :: p_1]]x & = & \mathcal{S}^{a_1}[[p_1]]x \\
\mathcal{S}^a[[n]]x & = & \{x_1 \mid x_1 \in \mathcal{A}[[a]]x, nodetype(x_1) = \mathcal{P}(a), \mathsf{name}(x_1) = n\} \\
\mathcal{S}^a[[*]]x & = & \{x_1 \mid x_1 \in \mathcal{A}[[a]]x, nodetype(x_1) = \mathcal{P}(a)\} \\
\mathcal{S}^a[[\mathsf{text}()]]x & = & \{x_1 \mid x_1 \in \mathcal{A}[[a]]x, nodetype(x_1) = \mathsf{Text}\} \\
\mathcal{S}^a[[p[q]]]x & = & \{x_1 \mid x_1 \in S_1, \mathcal{Q}[[q]](x_1, k, n)\} \text{ where} \\
 & & S_1 = \mathcal{S}^a[[p]]x, \quad n = size(S_1), \quad j = size(\{x_1 \mid x_1 \in S_1, x_1 \leq_{doc} x\}), \\
 & & k = j \text{ if } a \text{ is a forward axis, } k = n{+}1{-}j \text{ if } a \text{ is a reverse axis.}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{Q} & : & Qualifier \rightarrow (Node, Number, Number) \rightarrow Boolean \\
\mathcal{Q}[[q_1 \text{ and } q_2]](x,k,n) & = & \mathcal{Q}[[q_1]](x,k,n) \wedge \mathcal{Q}[[q_2]](x,k,n) \\
\mathcal{Q}[[q_1 \text{ or } q_2]](x,k,n) & = & \mathcal{Q}[[q_1]](x,k,n) \vee \mathcal{Q}[[q_2]](x,k,n) \\
\mathcal{Q}[[\text{not } q]](x,k,n) & = & \neg\mathcal{Q}[[q]](x,k,n) \\
\mathcal{Q}[[p]](x,k,n) & = & \mathcal{S}^{\mathsf{child}}[[p]](x) \\
\mathcal{Q}[[e_1 = e_2]](x,k,n) & = & \mathcal{E}[[e_1]](x,k,n) = \mathcal{E}[[e_2]](x,k,n)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{E} & : & Expr \rightarrow (Node, Number, Number) \rightarrow Number \\
\mathcal{E}[[e_1{+}e_2]](x,k,n) & = & \mathcal{E}[[e_1]](x,k,n){+}\mathcal{E}[[e_2]](x,k,n) \\
\mathcal{E}[[e_1 * e_2]](x,k,n) & = & \mathcal{E}[[e_1]](x,k,n) * \mathcal{E}[[e_2]](x,k,n) \\
\mathcal{E}[[\mathsf{position}()]](x,k,n) & = & k \\
\mathcal{E}[[\mathsf{last}()]](x,k,n) & = & n \\
\mathcal{E}[[i]](x,k,n) & = & i
\end{array}
$$

Figure 3.1: Formal Semantics of XPath according to [Wad99b]

3.2.2 XPath Weaknesses

Dereferencing. The core XML concept provides unidirectional intra-document references by `ID/IDREF/IDREFS` attributes. Using `IDREFS`, multi-target references can be specified. On the other hand, *using* references in XML querying languages is still problematic.

Example 3.4 (Dereferencing)
The following query is used throughout the paper for illustrating dereferencing:

> *"Select all names of cities which are seats of an organization and the capital of one of its members."*

In XPath, dereferencing is done via the id(...) function. Thus, navigation along several references becomes confusing:

id(//organization[id(./@seat) = id(id(./member/@country)/@capital)]/@seat)/@name

In contrast, in XML-QL [DFF+99b], dereferencing is done by joins (see Example 3.10).
Quilt [CRF00] (see also Section 3.10) adds a "→"-operator for explicit dereferencing. For the above query, the selection expression in Quilt is

//organization[@seat→= members/@country→/@capital]/@seat→/name/text() .

The KWEELT *[Sah00] implementation further specializes the syntax by adding* dereferencing hints *which specify where to search for the target of a reference: a dereferencing step is of the form*

refAttr→{elementtype$_1$@attr$_1$, ..., elementtype$_n$@attr$_n$}

The hint then means that the target of the reference is of one of the types elementtype$_1$, ..., elementtype$_n$ *where the attributes* attr$_1$, ..., attr$_n$*, respectively, must be checked. The above selection expression translates as*

//organization[@seat→{city@id} = members/@country→{country@car_code}/@capital]
/@seat→{city@id}/name/text() .

Since when using a DTD, it is sufficient to know which elementtypes are potential targets of the references (then the `ID` *attribute is uniquely defined by the DTD), XQuery [XQu01] (see Section 3.11) reduces the hint to a* name test *that specifies the expected name of the target element:*

//organization[@seat→city = members/@country→country/@capital]/@seat→city/name/text() .

In XPathLog, we will use implicit dereferencing: reference attributes are always *directly resolved, thus, we will write*

//organization[seat = member/@country/@capital]/@seat/name/text()

The resolving of `IDREF(S)` XQL and XML-QL is insufficient, see Sections 3.3 and 3.5.

Non-Resolving of `NMTOKENS` attributes. Whereas XPath handles the splitting and dereferencing of `IDREF(S)` by the id(...) operator, it is striking that, *even if a DTD is used*, XPath is *not aware* of `NMTOKENS` declarations:

- //country[@car_code="CH"]/@industry[1] = "machinery chemicals watches textiles",
  i.e., the `NMTOKENS` are not split,
- //organization[abbrev="EU"]/member/@country[1] = "A D I L NL ...",
  i.e., the `IDREFS` are not split, but
- id(//organization[abbrev="EU"]/member/@country)
  yields the country elements for Austria, Germany, Italy etc., i.e., when applying the id function; they are split, *and*
- if we add appropriate elements with id's "machinery", "chemicals", etc.,

  id(//country[@car_code="CH"]/@industry)

  yields these objects! – thus, `NMTOKENS` can also be "dereferenced" in XPath.

3.2.3 XPath as a Basic Concept

XPath is *not* an XML querying language, but only an addressing mechanism which selects node sets in XML documents. Compared with the relational algebra, especially a full join operator is missing (semi-equijoins are in fact provided by the path operator, and filters). Also, it cannot be used for *generating* XML trees. Its purpose is to provide the common addressing mechanism for XML, and to serve as a base for XML querying and manipulation languages and further concepts.

Several querying, transformation (which is a completely new concept, compared to the relational model), and manipulation languages have been presented up to now, providing the following functionality:

- Joins (XQL, XML-QL, Quilt/XQuery),

- Dereferencing,
  - via Joins (XQL, XML-QL, Quilt/XQuery),
  - via a dereferencing operator (Quilt/XQuery, YAXQL),
- Embedding in programming language constructs (XSLT, Tamino, eXcelon, YAXQL),
- Aggregation (Quilt/XQuery),
- Generation of XML trees (XSLT, XML-QL, Quilt/XQuery, YAXQL),
- Extension and usage of the addressing mechanism in XML: XPointer/XLink,
- XPath is also used as an addressing mechanism in XML Schema.

These languages are described in the sequel.

3.3 XQL

As mentioned above, *XQL (XML Query Language)* [RLS98, Rob99] was an early proposal for a simple querying language. The basic idea and syntax – using paths and filters for navigation – was the same as in XSL Patterns; although, the notions of *axis* and *location path* had not yet been defined. XQL only used the "/", "//" "/@" operators for navigating to children, descendants, and attributes – roughly, it is the fragment of XPath which can be built without using axis::; additionally, union and intersect on results were allowed.

The central extension, making XQL a *querying language* instead of a pure addressing mechanism, is to generate the result tree (instead of a nodeset) as a projection of the input document.

The 1998 XQL whitepaper [RLS98] solved this problem by *return operators* which may be placed after every prefix of an XQL expression. Return operators add parts of the current node to the output, generating a result tree:

?: the current node (i.e., its start tag, text contents, and end tag) is added to the result tree,

??: the whole current node is added to the result tree.

Example 3.5 (XQL Return Operators)
The following queries give an impression of XQL return operators:

- *country/city[@isCountryCapital]/name*

  <name>Berlin</name>
  <name>Rome</name>
  ⋮

- *country?/city[@isCountryCapital]/name*

  <country> <name>Berlin</name> </country>
  <country> <name>Rome</name> </country>
  ⋮

- *country?[@car_code?]/city[isCountryCap]/name*

  <country car_code="D"> <name>Berlin</name> </country>
  <country car_code="I"> <name>Rome</name> </country>
  ⋮

- *country?[@car_code?]/city?/name/text()*

```
<country car_code="D">
    <city>Berlin</city> <city>Munich</city> <city>Hamburg</city> ...
        ⋮
</country>
<country car_code="I">
    <city>Rome</city> <city>Milano</city> ...
        ⋮
</country>
    ⋮
```

The 1999 XQL proposal [RLS98,Rob99] replaced the return operators by *grouping*: The expression

$$path_1 \ \{ \ path_2 \ \}$$

returns all results x_1 of $path_1$, where the results of $path_2$ evaluating from x_1 as context node are nested inside:

Example 3.6 (XQL Grouping)
The XQL grouping mechanism generates nested results as follows:

```
country {@car_code | city {name/text()} }
```

returns

```
<country car_code="D">
    <city>Berlin</city> <city>Munich</city> <city>Hamburg</city> ...
        ⋮
</country>
<country car_code="I">
    <city>Rome</city> <city>Milano</city> ...
        ⋮
</country>
    ⋮
```

The same proposal also included functionality for querying and changing the order of subelements:

- The order of subelements can be queried by binary before and after predicates between nodes (*path*/a after b selects all a elements which occur after a b element). This functionality is covered by XPath's sibling axes.
- Sequencing allows for reordering of children in the output, e.g., *path*{a after b} (re)groups the inner elements such that first, all a elements are output, then all b elements.
- If no reordering is explicitly given, the elements retain their sequence in query results [RLS98].

All the above functionality, i.e., return operators, grouping, sequencing, and order-preserving have not been incorporated into XPath since XPath is designed as a pure addressing mechanism. Instead, the transformation/querying languages XSLT (W3C), XML-QL, and Quilt/XQuery incorporate restructuring mechanisms. An interesting aspect here is, that XPath does *not* specify the ordering of nodes in result sets.

Additionally, *asymmetric full joins* are provided via correlation variables. If an outer path contains a variable assignment *path*[$*var* := *expr*], the variable may be used as a join variable in an absolute path expression inside a group. These joins have also *restructuring* functionality, nesting elements which have not been nested before:

Example 3.7 (Join and Regrouping in XQL)
The following XQL query returns all continents with the name of the countries located on the continent:

```
//continent[$cont = id]
        { name, //country[encompassed/@continent = $cont]{ name/text() }}

<continent>
    <name>Europe</name>
    <country>Germany</country>
    <country>Italy</country>
        ⋮
</continent>
    ⋮
```

Handling reference attributes by joins over their values, XQL has a problem with `IDREFS` attributes: the attribute value is a sequence of ids, e.g., memberships="org-EU org-UN org-NATO ...". Joining this value with, e.g., <organization id="org-EU"> does not fit. Using string operations and the text containment operator, this functionality may be simulated, but in this aspect, the language is insufficient.

XQL influenced the design of XPath and subsequent concepts (see overview at the beginning of Section 3). XQL has been implemented in [HM99]. It served as querying language in the early versions of Tamino [Sof] and Excelon [eXc], where it now has been replaced by XPath.

3.4 XSL

In the early working drafts, XSL consisted of three components:

- XSL Patterns as an addressing and selection mechanism for XML trees (which was separated to XPath in July 1999).
- *XSL-FO (Formatting Objects; defining the namespace* fo:*)* which provides elements describing formatting and layout markup for XML documents (comparable to LaTeX's formatting environments and commands), e.g.,
  - page layout, areas, frames, indentation,
  - colors, fonts, sizes,
  - optical markup such as lists or tables, etc.

  These elements are added to the result tree when transforming XML documents (see below). XSL-FO-capable browsers are intended to translate these elements into HTML; other applications may translate them to LaTeX/RTF/PDF etc. Currently, the only XSL-FO-capable tool is FOP [Tau] (a Java program which translates XML documents to PDF).
- *XSLT (XSL Transformations; defining the namespace* xsl:*)* [XSL99] which serves as a functional-style programming language for transforming XML documents.

In contrast to XQL and XML-QL which are designed as querying languages (although XML-QL also allows for transformations using its nested queries functionality), XSLT is designed as a *transformation* language. XSLT stylesheets are applied to XML instances, yielding a new XML instance. Every XSLT stylesheet is a valid (wrt. the XSLT DTD) XML document, embedding XPath expressions as attribute values and text contents for addressing nodes. Stylesheets declaratively specify how elements should be manipulated. A stylesheet mainly consists of *templates* which specify

- to which elements the template applies, and
- what XML (sub)tree results from applying the template to a suitable element.

```
<xsl:template match="match-expr">
    contents
</xsl:template>
```

Here, *match-expr* is an XPath expression which is evaluated wrt. the root element. When the template is called for a context node, it is applicable to all elements which are addressed by *match-expr* (note that the selection of elements to which it is actually applied happens somewhere else). With this, an element may "run" several templates over all its subelements, handling each subelement by the suitable template. *contents* is a sequence of XSLT elements which contains the "program" for transforming selected nodes. It inserts nodes and text contents into the result tree:

- textual writing of values,
- copying nodes and values from the input tree:

  ```
  <xsl:template match="//country/province/city">
      <xsl:copy-of select="current()">
  </xsl:template>
  ```

  is a simple template which can be applied to *city* elements which are subelements of a *province* element which in course is a subelement of a *country* element, and copies them unchanged to the result tree,
- generating elements and attributes.

The actual application of elements is controlled by xsl:apply-templates elements (in the *contents* part of an xsl:template element):

```
<xsl:apply-templates select="xpath-expr"/>
```

Here, *xpath-expr* is an XPath expression which selects (wrt. the current context) the elements for which suitable (selected by the xsl:template match attribute) patterns should be applied:

- ```
  <xsl:apply-templates
      select="city[population > 1000000]"/>
  ```

  handles all *city* subelements of the current context node whose text contents of their *population* subelement is > 1000000. Note that this command does not specify *which* template is actually applied (the template whose match attribute matches the city element will be applied – for conflict resolving strategies, see [XSL99]).
- ```
  <xsl:apply-templates select="id(@capital)"/>
  ```

  handles the element in the current XML document whose id equals the value of the *capital* (reference) attribute of the current context node.

Example 3.8 (XSLT Stylesheet)
The stylesheet given below generates a result tree which first lists all country *elements, and then all* city *elements:*

```
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<!-- define template which simply copies country and city elements to the result tree -->
<xsl:template
      match="city|country">
      <xsl:copy-of select="current()"/>
</xsl:template>
<template match="mondial">
      <!-- apply template: countries -->
      <xsl:apply-templates select="/mondial/country">
      <!-- ... and then cities -->
      <xsl:apply-templates select="//country/city | //country/province/city"/>
</ template>
</xsl:stylesheet>
```

Named templates can be defined by <xsl:call-template name="*name*"/> elements.

Generating the Result Tree. The execution of an XSLT stylesheet on an XML instance starts with applying the template which matches the outermost element of the instance. Then, recursively, the processing is controlled by <xsl:apply-templates> and <xsl:call-template> where the template executions contribute to the result tree generation:

- all tags, complete elements and attributes in the *contents* of templates which do not belong to the xsl: namespace (i.e., which are not XSLT commands) are added verbatim to the result tree.
- Nodes are copied from the input tree to the result tree by <xsl:copy-of select="*xpath-expr*"/>; literal values are copied by <xsl:value-of select="*xpath-expr*"/>.
- For assigning attribute values, the value may be specified by an XPath expression, e.g.

  ```
  <elementname attribute="xpath-expr">
  ```

- New elements can be generated from scratch using

  ```
  <xsl:element name="xpath-expr">
     <xsl:attribute name="xpath-expr">
        contents
     </xsl:attribute>
  </xsl:element>
  ```

  Here, the *names* can be computed by *xpath-expr*; the actual element contents is computed by recursively applying the XSLT commands in *contents*.

XSLT also provides the common procedural concepts (loops, conditions) for controlling the processing of nodes.

- An important operation when processing a document is the iteration over the result set of an XPath expression (evaluated wrt. the current context node):

  ```
  <xsl:for-each select="xpath-expr">
        contents
  </xsl:for-each>
  ```

- conditional execution is provided by test or case-splits:

<xsl:if test="*predicate*"> *contents* </xsl:if>

<xsl:choose>
<xsl:when test="*predicate*$_1$"> *contents*$_1$ </xsl:when>
<xsl:when test="*predicate*$_2$"> *contents*$_2$ </xsl:when>
⋮
<xsl:otherwise> *contents*$_{n+1}$ </xsl:otherwise>
</xsl:choose>

XSLT variables can be defined and once assigned with a value (forest, node or literal) by

<xsl:variable name="*var-name*" select="*xpath-expr*">
contents
</xsl:variable>

where the value is *either* given by the select attribute, or by the *contents* (which allows for constructing subtrees). Variables are then used either by

<*elementname attribute*="$*var-name*"/>

for defining attribute values, or by elements of the form

<xsl:value-of select="$*var-name*"/>

which converts the variable's value into a string (for use e.g. as attribute value), or by

<xsl:copy-of select="$*var-name*"/>

which adds the variable contents to the result tree.

Communication of variables between templates is done by *parameterized templates*. Parameters are declared for templates by

<xsl:template match="...">
<xsl:param name="*param-name*" select="*xpath-expr*/">
</xsl:template>

When the template is called, the parameter value is given as an attribute:

<xsl:apply-templates select="*xpath-expr*$_1$">
<xsl:with-param name="*param-name*" select="*xpath-expr*$_2$"/>
</xsl:apply-templates>

XSLT follows the LISP concept, where the program itself can also be regarded as data in the data model which underlies the language. This allows also to handle *programs* with XSLT, e.g., rewrite them, or to exchange programs as XML data, identify relevant subqueries, and reuse them. This aspect is again considered in the next section where the W3C activities for designing *the* XML querying language are described.

One of the main drawbacks of XSLT as a database language (it is not intended as a database language, but as a transformation language) is that it is not possible to *update* the document or the result tree. For data integration, it is often necessary first to generate a skeleton of the result, and then refine it stepwise by adding information items from the individual sources (see Section 11).

Since XSLT allows to add nodes (i.e., subtrees) by (deep)-copying to the result tree, the result tree may contain dangling reference attributes (i.e., reference whose target is not present in the result tree).

There are multiple implementations; the development phase has been accompanied by XT [Cla98] which has been the most efficient implementation up to 2001. XT was employed as XSLT tool in this work. Saxon [Kay99] is another implementation of XSLT (building on the event-based SAX API) since the early days, also providing several innovative extensions in the saxon: namespace. Of course, commercial implementations are also available. Different implementations provide (yet proprietary) extensions:

- Saxon and XT provide the *saxon:nodeset()* and *xt:nodeset()* function which allows to access the result tree generated so far as an input source.
- eXcelon [eXc] – which is a commercial XML database system – extends XSLT with an update language *XUL (XML Update Language)* which allows to specify updates on elements which are selected by XPath expressions.

For accessing several documents, several tools provide the proprietary function document(*url*) for accessing the document residing at *url* as starting point for XPath expressions.

3.5 XML-QL

XML-QL [DFF+98, DFF+99b, DFF+99a] is another early (1998, non-W3C) proposal for an *XML querying and transformation* language. The design (and implementation) of XML-QL has been influenced by STRUDEL/STRUQL [FFLS97, FFK+98]. In contrast to XSL Patterns/XQL, XML-QL does not employ navigation and paths, but XML patterns which are matched against the queried document. The basic idea was influenced by SQL-like languages, partitioning XML-QL queries into a *selection part* (`WHERE ... IN`) and a *construction part* (`CONSTRUCT`):

WHERE *xml-pattern*$_1$
IN *url*
CONSTRUCT *xml-pattern*$_2$

Here, *xml-pattern*$_1$ is matched against the XML instance given by *url*. Every match yields variable bindings which are

- used as join variables, and
- propagated to the result,

which is again an XML pattern specifying the result tree:

Example 3.9 (XML-QL)
The following XML-QL query transforms the name *subelements of* countries *into attributes. The* ***WHERE*** *part is matched against the document "www.../mondial.xml" where it matches all* country *elements and assigns the variables* ***$id*** *and* ***$name*** *to the value of the* car_code *attribute and the text contents of the* name *element, respectively:*

```
WHERE
    <country car_code=$id>
        <name>$name</>
    </>
    IN "www.../mondial.xml"
CONSTRUCT <country car_code=$id name=$name> </>
```

For every variable binding, the ***CONSTRUCT*** *clause generates a new* country *element in the result:*

```
<result>
  <country car_code="D" name="Germany"> </>
  <country car_code="F" name="France"> </>
    ⋮
</>
```

The WHERE part can be extended to multiple patterns (also applicable to multiple documents) by

WHERE ($expr_1$ IN doc_1, ... , $expr_n$ IN doc_n)

Here, variables which occur in several patterns act as join variables, which is especially required for evaluating references:

Example 3.10 (XML-QL: Dereferencing via Join)
The following query computes all pairs of countries c and organizations o s.t. c is a member of o:

```
WHERE
  <country car_code=$c> </>
  IN mondial.xml,
  <organization abbrev=$org>
    <membership type=$type country=$c> </>
  </>
  IN mondial.xml
CONSTRUCT
    <ismember country=$c organization=$org type=$type> </>
```

Similar, multiple sources can be accessed:

```
WHERE
  <city name=$name2> </>
  IN www.../europe.xml,
  <city name=$name1> </>
  IN www.../america.xml,
  <connection from=$name1 to=$name2> </>
  IN www.../lufthansa.xml
CONSTRUCT
  <connection>
    <from continent="europe" city= $name1> </>
    <to continent="america" city= $name2> </>
  </>
```

The IN part may also refer to the contents of a variable defined in a previous WHERE part. In fact, this implements the semijoin semantics of XPath filters:

Example 3.11 (Filtering in XML-QL)
The following XML-QL expression selects all elements which have a name *subelement with text contents "Monaco":*

```
WHERE
  $element
  IN mondial.xml,
  <name>Monaco</>
  IN $element
CONSTRUCT
  $element
```

Similar to SQL, the `CONSTRUCT` part may contain nested XML-QL Queries. Here, the `IN` part applies

- either to an url (if the inner query on *another* document is correlated with the contents of the current element), or
- to the contents of a variable which has been assigned in the `WHERE` part (if the inner query is used to restructure the *contents* of the current element).

Both cases are illustrated by the following examples.

Example 3.12 (Result Grouping in XML-QL)
Nested queries are required when the structure of a document has to be changed and elements are grouped. The following does the same as the XQL query in Example 3.7:

```
WHERE <continent> <name>$cont </> </>
    IN mondial.xml,
CONSTRUCT
    <continent> <name>$cont </>
    WHERE <country >
                <name>$name </>
                <encompassed continent=$cont></>
            </>
    IN mondial.xml
    CONSTRUCT <country>$name</>
    </>
```

Nested queries using a variable in the `IN` statement are useful if the contents of the current element should be restructured:

Example 3.13 (XML-QL: Nested Queries)
The following reduces the nested city *descendant subelements of* country *elements to their names:*

```
WHERE <country> $country</>
    IN mondial.xml,
CONSTRUCT
    <country>
    WHERE <name >$name </>
    IN $country
    CONSTRUCT <name>$name</>
    WHERE <city >
                <name>$name </>
            </>
    IN $country
    CONSTRUCT <city>$name</>
    </>
```

generates

```
<country>
    <name>Germany</name>
    <city>Berlin</city>
    <city>Hamburg</city>
        ⋮
</country>
    ⋮
```

In addition to the XML-QL language, [DFF+99b] defines a *graph-based data model* related to the one defined in [Bun97]. In contrast to the XML data model which is incorporated into the DOM model and into the XML Query Data Model [XMQ01b] (see also Section 3.9.1), the graph is *edge-labeled* and supports implicit dereferencing. The data model has been influenced by the experiences with the Strudel/StruQL [FFK+98] project.

Although, the XML-Pattern syntax of XML-QL is not extensible in this way. The syntax given in [DFF+99b] solves the problem by identifying reference attributes with elements, e.g., the XML-QL pattern

```
WHERE <country> <capital name = $name> </> </>
IN mondial.xml
```

would have the same semantics as XPath's id(//country/capital)/@name. In case that there is also a *capital* subelement, the XML-QL syntax is not well-defined. This solution is – at least – questionable. Since joins are supported in XML-QL, the expression

```
WHERE <country capital = $cap>
                <name>$cname</></>
          IN mondial.xml,
          <city id = $cap>
                <name>$capname</></>
          IN mondial.xml
CONSTRUCT ...
```

is preferable.

Similar to XQL, XML-QLs handling of `IDREFS` attributes in the querying/matching part is insufficient since references are handled by joins or nested subqueries (thus, also by joins): The value of an `IDREFS` attribute is a sequence of ids, e.g., memberships="org-EU org-UN org-NATO ..." which does not match with a single id value, e.g., "org-EU". Considering the fact that the *same* paper also defines the above-mentioned *graph-based data model*, adding reference cross edges to the (edge-labeled) XML tree, it is surprising that the consequences have not been dealt with when designing XML-QL.

The `WHERE` clause provides some constructs for switching between elements and contents, e.g.,

```
WHERE <country>
        <name>$n </>
        <city>$c </>ELEMENT_ AS $x
        </>
    IN mondial.xml,
CONSTRUCT ...
```

binds $x to <city> ... </city> elements. Analogously for CONTENT_AS.

A kind of *element fusion* (cf. Section 11.6) is supported by *skolem functions* which can e.g., be used for specifying outer joins over several sources:

Example 3.14 (Element Fusion)
The expression

```
{ WHERE <country> <name>$n </>... </>IN cia.xml
    CONSTRUCT <country ID="c($n)"><name>$n </> ... </>
} { WHERE <country> <name>$n </>... </>IN gs.xml
    CONSTRUCT <country ID="c($n)"><name>$n </> ... </>
}
```

collects countries from the sources cia *and* gs*. When a country occurs in both of them (by name), the two elements are combined.*

Advanced features in XML-QL include *tag variables* and *regular path expressions* similar as defined in the LOREL language over the OEM model in [GMPQ+97, AQM+97a, GMW99]. Compared to XQL, XML-QL provides much more transformation functionality. Although, compared to XSLT, the transformation functionality is restricted: Complex transformations are hard to implement, using deeply nested filtering (i.e., iterative `WHERE` clauses) and complex `CONSTRUCT` clauses. Similar to SQL, XML-QL does not support recursive queries.

Using XML patterns, XML-QL has no notion of the XML *axes* except the child and attribute axis, i.e., selecting nodes amongst siblings or ancestors. The descendant axis is supported by regular path expressions: e.g., *path*/country/descendant::city can be encoded by the pattern

.../ <country> <(any)*> <city> ... </> </> </> .

As already stated above, XML-QL uses a data model different from the W3C XML Query Data Model, influenced by the object-oriented data model of the STRUDEL/STRUQL [FFK+98] project. There is an ordered data model (for documents), and an unordered data model (for databases); for an investigation of this duality see also Section 4. The unordered data model is a graph with a distinguished *root* node in which each node has a unique *object identifier*. The graph is labeled as follows: the edges are labeled with *element tags*, the nodes are labeled with sets of attribute-value pairs, and the leaves (text nodes) are labeled with string values. The ordered model extends the unordered one by associating an order of the *complete* set of nodes (i.e., the order of the outgoing edges (i.e., children) of each node is induced by this global order).

A very similar graph-based model (but with local order) is used in our approach, also influenced by the experiences with research on semistructured data in an object-oriented model. For a comparison, see Section 8.3.

Similar to XSLT, since XML-QL allows to bind variables to nodes (i.e., subtrees) and (deep)-copy them into the result tree, the result tree may contain dangling reference attributes.

XML-QL has been implemented in [DFF+99c].

3.6 XML Query Requirements

Based on the experiences with XSL Patterns/XQL, XSLT, and XML-QL, the requirements on an XML querying language have been stated by in the *W3C Query Requirements* working draft [XMQ01c]:

- The prospective XML querying language must cover the aspects of both *document-oriented* (human readable) and *data-oriented* (XML as an electronic data exchange format) documents,
- it must be declarative, not enforcing a particular evaluation strategy,
- it must support the XML 1.0 datatypes, and the *complexTypes* definable in XML Schema,
- it must support references, both within a document, and from one XML instance to another (i.e., navigation along XLinks in queries; for an investigation on this topic see Section 13),
- it should have several language bindings. One language syntax must be convenient for humans, and *one syntax must be expressed in XML*. The idea here is – similar to XSLT – that queries are XML instances and thus can be manipulated themselves by XML languages.

An important aspect is the design of the XML world as a collection of related standards (document format, querying language, metadata language, web-oriented features such as linking and application-oriented extensions) based on the XPath addressing mechanism, and expressed themselves as XML trees. The outline of this picture has been completed by that time with complementary laguages, e.g., XML Schema (see Section 3.7) for describing metadata, and XLink (see Section 3.8) for specifying links between XML instances. We first describe these two concepts, and then continue with XML querying in Section 3.9.

3.7 XML Schema

XML Schema instances are XML trees, built from elements and tags from a special namespace with a predefined semantics. The concepts correspond to those of schema specifications in relational and object-oriented databases. From the database point of view, XML Schemas are preferable against DTDs due to their modeling concepts. The XML Schema Working Draft [XML99a] consists of two parts:

- definition of datatypes,
- definition of structures (complex datatypes, elements, and attributes).

An XML Schema instance which describes the MONDIAL database can be found in Appendix B.

Data Model. In contrast to DTDs which are a heritage from the SGML document area, the data model behind XML Schema has been influenced by the database community. An important difference to DTDs is that XML Schema distinguishes *datatypes* (including *complexTypes* which act as element types) from *element names* and allows for a much more detailed specification of *complexTypes*.

Datatypes. The definition of datatypes allows for a very detailed specification, given as 3-tuples consisting of (using the terminology of [XML99c])

- a *value space* (domain) which may be given intensionally (by axioms), extensionally (enumeration), derived from another domain, or by construction from several other domains (e.g., constructing tuples or sets),
- a *lexical space* (the lexical representations of its elements, e.g., 100 = 1.0E2 = 1.0e+2),
- *facets* which characterize properties of its elements (equality, order, bounds etc). An important facet is *numeric*.

XML Schema uses a set of predefined primitive and generated datatypes, e.g., string (which has a subtype `NMTOKEN`), boolean, float, double, decimal (with subtypes integer etc.), uriReference (Universal Resource Identifier), timeDuration, recurringDuration etc. which are roughly the same as for SQL. For the use as attributes, there are several XML-specific datatypes which are defined according to the XML standard, e.g., `NMTOKEN(S)`, `Name`, `QName` etc.

Additionally, XML Schema allows to use datatypes for attributes whose extension depends on the given document, e.g., `ID` and `IDREF(S)` which are relevant in this work. Note that lists such as `NMTOKENS` and `IDREFS` are treated as atomic datatypes.

Contents of XML Schema documents.
The contents of an XMLSchema documents consists of

- type definitions,
- attribute declarations, and
- element declarations.

User-defined datatypes can be derived from existing datatypes as <simpleType> (defining datatypes for attribute values of text contents) or complexType (defining datatypes for elements),

- as a list of some base type (cf. collections in the ODMG standard for object-oriented databases [CB00]), or
- by restriction of a base type (using restricting facets).

Thus, the type definitions induce a *type definition hierarchy* of datatypes.

Simple Types. *Simple types* are "literal" types which are used for attribute values of text contents. Thus the main interest when deriving simple types is to express range restrictions.

Example 3.15 (XML Schema: Derived Simple Types)
The following XML schema elements derive restricted atomic datatypes from float *for* longitude *and* latitude*:*

```
<simpleType name="longitude" base="float">
        <minExclusive>-180</minExclusive>
        <maxInclusive>180</maxInclusive>
</simpleType>

<simpleType name="latitude" base="float">
        <minInclusive>-90</minInclusive>
        <maxInclusive>90</maxInclusive>
</simpleType>
```

Complex Types. Complex datatypes can be derived from existing datatypes using <complexType> elements. They may be defined by restricting another datatype (either in its components, or in its structure), or by extending a given one, or creating a completely new one. These datatypes are then used for defining *element types*. This is one of the main differences to DTDs: the semantical notion, i.e., *elements* – roughly spoken, the tags which are then used in the document – are different from the structural notion, i.e., datatypes (defined as *complexTypes*). Complex types

- can be derived from a *simpleType* (which then defines the text contents),
- can be derived from a *complexType* (which then defines text contents, some children, and some attributes),
- define additional content model particles (appended to the content model of the type which is extended),
- define additional attributes.

For an investigation of *structural inheritance* when deriving a *complexType* from another *complexType*, see Section 10.3. A *complexType* definition is specified by the following properties:

- name,
- base type (which may be a *simpleType* or a *complexType*),
- derivation method (*extension* or *restriction*),
- attribute declarations,
- content type (*elementOnly*, *empty*, *mixed*, or *simpleType* (which means, *characterOnly* contents)),
- contents model (in case of *elementOnly*).

```
<complexType name="name"
             base="basetype"
             content="elementOnly|empty|mixed|textOnly"
             derivedBy="extension|restriction" >
    element-declarations or facets
    attribute-declarations
</complexType>
```

The element contents specifies attributes and content model of the *complexType* to be defined:

- if *basetype* is a complex or a simple type, and *derivedBy*="restriction", it contains facets which restrict the *basetype*;
- otherwise it contains <element> and <attribute> subelements as described below which declare attributes and the structure of element contents. The element contents may further be nested into <all>, <choice>, <sequence>, and <any> elements which allow for specifying similar properties as for DTDs. For a pure database view, the content is simply specified by a sequence of element subelements.

Attribute definitions and element definitions are allowed both globally, i.e., as immediate children of the <schema> element, then they may be referenced from arbitrary <complexType> elements (see below), or locally inside a <complexType> element, then they are local.

Attribute Definitions. Attribute definitions are given by <attribute> elements which specify

- name,
- type (a *simpleType* which may be defined at that place),
- minimal (optional vs. required) and maximal cardinality,
- default value, or a fixed value.

The syntax of the global form is

```
<attribute name="name" type="simpleType" value="..." />
```

Recall that *simpleType* includes `NMTOKENS` and `IDREFS`.
In the local form, <attribute> elements occur inside <complexType> elements. Then, attribute declarations are of the form as above, or refer to global ones:

```
<complexType name="name"
             base="basetype" >
  element-declarations
  <attribute ref="global_attribute_name"
           use="default|fixed|optional|prohibited|required"
           value="..."/>
  ⋮
</complexType>
```

The default of *use* is *optional*, and the declaration of *value* can be inherited from *global_attribute_name*. *maxOccurs* is `FIXED`="1" since there is *one* attribute of this name – recall that `NMTOKENS` and `IDREFS` are base types.

Element Definitions. Element definitions are given by <element> elements which specify

- name,
- type, which may be a *simpleType* or a *complexType*; also local type definitions are allowed. Note that using a *simpleType* in fact specifies the type of the *element contents* and tells that the elements should behave as literals in certain situations (see Example 3.16; this aspect is again analyzed later in Section 5.6),
- default value, fixed value,
- integrity constraints (keys, foreign key references similar to SQL)

The syntax of the global form is as follows:

```
<element name="name" type="typename" default="..." fixed="...">
  integrity-constraints
</element>
```

Here, a default/fixed value is allowed only for elements which contain any text contents (note that a default for *element contents* is not possible for DTDs).

Complex types cont'd. The definition of complex types uses the above <attribute> and <element> elements. The attributes are simply declared by a list of <attribute> elements (there is no order of attributes). Since the content model allows complex definitions of sequence, choice, etc., here, additional elements for constraining the structure are provided. The simplest form is a sequence of <element> elements which does not impose any restrictions on the order of subelements. Several kinds of definitions of subelements are possible:

- referring to global element definitions,
- local element definition, referring to global types, and
- local element definitions, using a locally defined type.

```
<complexType name="name" ... >
   attribute-declarations
   <element ref="element-name"> maxOccurs="..." minOccurs="..."/>
   <element name="name" type="typename"
             default="..." fixed="..."
             maxOccurs="..." minOccurs="..." >
             integrity-constraints
   </element>
   <element name="name"
     default="..." fixed="..."
     maxOccurs="..." minOccurs="..." >
     type-definition
     integrity-constraints
   </element>
     ⋮
</complexType>
```

where *element-name* is the name of a global element declaration. If a local type is defined, it gets no name.

ComplexType declarations define local *symbol spaces*, i.e., the same element name may be used in several complex data types with different element types (which was not possible in DTDs).

Example 3.16 (XML Schema: Complex Types)
In the MONDIAL *database, the* population *subelements of* country *and* city *are different (the full XML Schema instance is given in Appendix B):*

- city/population *has a year attribute,*
- country/population *does not.*

```
<complexType name="country">
  <attribute name="car_code" ... />
  <element name="population" type="integer" ... />
```

```
        ⋮
</complexType name="country">

<complexType name="city">
    <element ref="name"/>
        ⋮
    <element name="population" ... >
        <complexType base="integer" derivedBy="extension">
            <attribute name="year" type ="date" use="optional"/>
        </complexType>
    </element>
        ⋮
</complexType>
```

An excerpt of the database, containing both types of population elements is e.g.,

```
<country car_code="CH" capital="cty-Bern" memberships="org-efta org-un ...">
    <name>Switzerland</name>
    <population>7207060</population>
    <city id="cty-Bern" name="Bern">
        <population year="91">134393</population>
    </city>
        ⋮
</country>
```

This example shows another interesting property: both types are derived from the simpleType integer*, thus, they are sometimes expected to act like integers, e.g., in comparisons: the XPath query*

```
//country[population > 5000000]/name/text()
```

returns "Switzerland" in its result set. This problem is investigated in Section 5.6.

Integrity Constraints. XML Schema supports *identity constraints* and *referential integrity constraints* known from the relational model:

- unique
- key
- keyref

Here, *unique/key* and *keyref* are more expressive than the ID/IDREF concept: *unique/key* specifies a list of properties which must uniquely identify each item amongst a set of nodes which are addressed by a *selector* (the selector is again an XPath relative location path). By nesting <key> elements *inside* <complexType> elements, even *local* keys and foreign keys can be defined. Here, only the semantics of a simple form of *global* keys is described which corresponds to the *key* concept in SQL:
The following *key* element declares the tuple

$$(x.xpath\text{-}expr_1, \ldots, x.xpath\text{-}expr_n)$$

to be the key of x amongst all elements x addressed by //*name*:

```
<key name="name" >
      <selector>//name</selector>
      <field>xpath-expr1</field>
         ⋮
      <field>xpath-exprn</field>
</key>
```

where every *xpath-expr*$_i$ must have a unique value for each element addressed by *xpath-expr*.
In contrast to the ID/IDREF semantics,

- keys are not unique to the document, but only to a specified domain (above, all elements addressed by //*name*), and
- keys may be composite and may also include element contents (including text contents).

Similarly, <keyref> elements relate a set of *referencing nodes* to *referenced nodes* by specifying a *foreign key* which references a certain *key*:

```
<keyref name="..."   refer="keyname" >
    <selector>//name</selector>
    <field>xpath-expr1</field>
    ...
    <field>xpath-expr1</field>
</keyref>
```

The above <keyref> specifies that for all elements selected by evaluating //*name*, the (unique) values selected by

$(x/\textit{xpath-expr}_1, \ldots, x/\textit{xpath-expr}_n)$

must reference the key which is specified by the corresponding <key name="*keyname*"> element.

Example 3.17 (XML Schema: Referential Integrity)
The following excerpt of an the MONDIAL *XML Schema declares the* id *attribute as the key of* city *elements (<key name="citykey">), and declares the* capital *attribute of* country *elements as a foreign key, referencing* citykey*:*

```
<complexType name="country">
      <attribute name="car_code" type="ID" .../>
      <attribute name="capital" type="IDREF" .../>
         ⋮
      <element name="city" type="city" .../>
         ⋮
</complexType>
<complexType name="city">
      <attribute name="id" type="ID" use="required"/>
         ⋮
</complexType>
<key name="citykey">
      <selector>//city</selector>
      <field>@id</field>
```

```
</key>

<keyref name="country2capital" refer="citykey" >
    <selector>//country</selector>
    <field>@capital</field>
</keyref>
```

As in the above example, key and keyref constructs can be used for specifiying the target elements of `IDREF` attributes. Note that this does *not* hold for `IDREFS` since the value of an `IDREFS` attribute is a *list* of keys.

3.8 XPointer and XLink

XPointer [XPt00] is a specialized extension of XPath for selecting parts of XML documents – which are not necessarily sets of nodes: every "area" which can be selected by clicking and marking can be described by an XPointer. For this work, only pointers in the sense of "inter-database" references documents are relevant. Thus, the investigation is restricted to pointers which select node sets – i.e., we consider only XPath expressions as pointers.

The simplest kind of pointers is already provided by the <A> tag known from HTML: By <A HREF="*url*">, a link to a given url can be specified (with the semantics that clicking on it, the browser accesses the referenced url). HTML also allows an extension by *anchors*: if the target document contains an *anchor* specified by <A NAME="*name*">, a hyperlink to this place can be defined by <A HREF="*url#name*">. Note that this only works, if the author of the source has defined a suitable anchor.

The XPointer concept extends the URL and HTML-hyperlink concepts: The URL addressing mechanism is combined with XPath, and extended for addressing sections or nodesets inside the target document: XPointers are of the form

XPointer = *url#(extended)-xpath-expr*

E.g., the following XPointer addresses the *country* element which has a *car_code* attribute with value "D" in the document with the url `www.our.server.de/Mondial/mondial.xml`:

www.our.server.de/Mondial/mondial.xml#descendant::country[@car_code="D"]

An important aspect is that this addressing does not depend on prepared *anchors* in the target document, as it is the case for HTML. Thus, everybody can define XLinks from his XML documents to any other XML source.

XPointers are required to be transparent against *mechanical* changes (i.e., inserting linebreaks or indentation for formatting an XML source) in the referenced document. In case that the ids used in the target document are known, the id() function provides a robust way for addressing elements – it is even stable wrt. restructuring of the document. Additionally, XML servers can maintain indexes for efficient access via id.

XML Linking Language (XLink). XPointers are used in the *XML Linking Language (XLink)* [XLi00] for – as the name says – expressing links between XML documents. The syntax and semantics of XLink is defined via XLL (Extensible Linking Language), using the xlink: namespace. The xlink: namespace provides certain element types with certain attributes with a standardized semantics which defines the behavior of applications and XML servers.

HTML Hyperlinks. HTML Hyperlinks are specified in the source document, addressing a specific point in the target document which has to be prepared by its owner. They are unidirectional, i.e., it is neither possible to check if somebody uses a certain anchor, nor to navigate from the anchor to any document using it. Additionally, each hyperlink has only one target, it is not possible to associate a set of targets with a single hyperlink.

References inside an XML document. The core XML concept already provides unidirectional references by `ID/IDREF/IDREFS` attributes. Using `IDREFS`, multi-target references can be specified. Here, dereferencing is done as shown above with the id(..) operator in XPath (or by pointer-chasing as in Quilt/XQuery). Note that by id(*xpath–expr*), a set of nodes can be addressed if

- *xpath-expr* is of the form *path*/@*attr* and the result set of *path* contains several nodes, or
- any of the attribute nodes in the result set of *xpath-expr* is of type `IDREFS` and has more than one entry.[3]

Thus the XML referencing mechanism is *unidirectional*, already *many-valued*, but *intra-document*. XLink provides several additional linking types:

- inter-document references; even a single reference can have targets in several documents,
- so far, the links are *inline* links, i.e., the link is directly contained in the document.
- XLink also provides *out-of-line* links which allow users to create a private collection *between* public documents.
- out-of-line links allow for bidirectional references – which are different from two complementary unidirectional references.

In all cases, the applications have to care about implementing the intended semantics specified for XLL. XLL also allows for specifying the *behavior* to be executed when a link is activated. Whereas the HTML behavior is simply "browse this document", XLL allows for more sophisticated activities (e.g., not to delay the behavior until the link is activated by a user, but evaluate it immediately when the document is parsed). For an adaptation to the requirements when querying XLinks in databases, see Section 13.

XLinks. Arbitrary elements can be declared to have XLink functionality (to choose from the functionality predefined in XLL) by equipping them with an xlink:type attribute and suitable additional attributes and subelements from the xlink: namespace:

```
<!ELEMENT linkelement (contentsmodel)>
<!ATTLIST linkelement
      xmlns:xlink   CDATA  #FIXED  "http://www.w3.org/xml/xlink/0.9"
      xlink:type    (simple|extended|locator|arc)  #FIXED   "..."
      ... >
```

The xlink:type attribute selects between four basic types of XLink functionality:

- simple: roughly, the functionality known from <A href="...">,
- extended: the same, allowing for multiple targets,
- locator: a special element type which is used for specifying targets which are then used by extended link elements,
- arc: another special element type which is used for defining bidirectional links between a set of locator elements.

The information that some element type is equipped with link semantics is given in the DTD: there, the attributes of the element type are declared, including attributes from the xlink: namespace. The xlink:type attribute is assigned with a fixed value.

[3] even attributes declared as `NMTOKEN` or `NMTOKENS` can be (mis)used for dereferencing.

```
<!ELEMENT linkelement (contentsmodel)>
    % Dependent on the xlink:type, some conditions on
    % the contentsmodel apply.
<!ATTLIST linkelement
        xmlns:xlink   CDATA  #FIXED "http://www.w3.org/xml/xlink/0.9"
        xlink:type    (simple|extended|locator|arc)  #FIXED "..."
                      % specifies the link element type
        xlink:href    CDATA  #REQUIRED
        xlink:title   CDATA  #IMPLIED
        xlink:role    CDATA  #IMPLIED
        xlink:show    (new|parsed|replace)  #IMPLIED
        xlink:actuate (auto|user)   #IMPLIED >
```

The other xlink:-specific attributes may also be specified with default or fixed values, or defined for each individual instance. They have the following semantics:

- xlink:href: selects a target for the individual instance (it "is" the link),
- xlink:actuate: defines the event on which the link is activated:
  - auto: the link is activated when the link element node is parsed,
  - user: the link is activated on user interaction (e.g., clicking).
- xlink:show: specifies what happens if the link is activated (currently defined only for browsing applications specifying if the target is replaces the current document, or if it is opened in a new window),
- xlink:role and xlink:title: an application-specific characterization of the link.

Example 3.18 (Distributed Mondial)
In the following, XLinks are introduced for a "distributed" version of MONDIAL *where all countries, all cities of a country, all organizations, and all membership relations are stored in separate files*

- *mondial-countries.xml (all organizations)*
- *mondial-cities-of-car-code.xml (the cities for each country)*
- *mondial-organizations.xml (all organizations)*
- *mondial-memberships.xml (relates countries and organizations)*

Simple Links. A simple link is similar to the HTML <A href="..."> construct.

Example 3.19 (Simple Links)
The @seat *attribute of organizations is replaced by a* seat *subelement which is a simple link:*

```
<!ELEMENT organization (...seat ...)>
<!ELEMENT seat EMPTY>
<!ATTLIST seat  xmlns:xlink   CDATA #FIXED "http://www.w3.org/xml/xlink/0.9"
                xlink:type    (simple|extended|locator|arc)  #FIXED "simple"
                xlink:href    CDATA #REQUIRED
                xlink:title   CDATA #IMPLIED
                xlink:role    CDATA #FIXED  "seat">
```

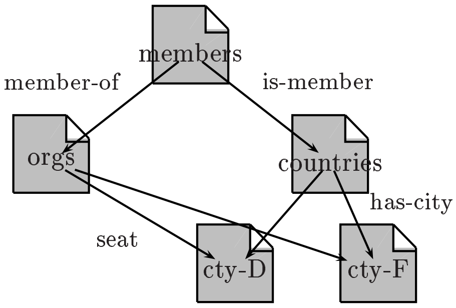

Figure 3.2: Distributed XML MONDIAL Database

An excerpt of the XML document looks as follows:

```
<organization id="org-UN">
   <seat href="file:cities-USA.xml#//city[name/text()='New York']"/>
        ⋮
</organization>
```

Inline extended Links. Extended links do not have an xlink:href attribute, but instead they contain one or more locator elements, each of which is a reference of its own.

```
<!ELEMENT linkelement (... locatorelement* ... )>
<!ATTLIST linkelement
      xmlns:xlink  ...        ...
      xlink:type   (simple|extended|locator|arc)  #FIXED "extended"
      xlink:title  CDATA  #IMPLIED
      xlink:role   CDATA  #IMPLIED  >

<!ELEMENT locatorelement (contentsmodel)>
<!ATTLIST locatorelement
      xmlns:xlink  ...        ...
      xlink:type   (simple|extended|locator|arc)  #FIXED "locator"
      xlink:href   CDATA  #REQUIRED
      id           ID     #REQUIRED
      xlink:title  CDATA  #IMPLIED
      xlink:role   CDATA  #IMPLIED  >
```

Example 3.20 (Extended, Multi-Target Links)
In the distributed MONDIAL *model, every* country *element has an extended, multi-target link element* cities *which contains* <*city*> *locators to all its cities:*

```
<!ELEMENT country (...cities ...)>

<!ELEMENT cities (city*)>
<!ATTLIST cities  xmlns:xlink   ...        ...
                  xlink:type    (simple|extended|locator|arc)  #FIXED "extended"
                  xlink:title   CDATA  #IMPLIED
                  xlink:role    CDATA  #FIXED  "cities" >

<!ELEMENT city (#PCDATA)>
<!ATTLIST city  xmlns:xlink   ...        ...
```

```
xlink:type    (simple|extended|locator|arc)  #FIXED  “locator”
xlink:href    CDATA  #REQUIRED
id            ID     #REQUIRED
xlink:title   CDATA  #IMPLIED
xlink:role    CDATA  #IMPLIED  >
```

Note that the xlink:role *attribute provides a possibility for giving additional information, e.g., if the city is the capital, a province capital etc. An excerpt of the XML document* ***mondial-countries.xml*** *may look as follows:*

```
<country car_code=“USA”>
   <cities>
      <city href=“file:cities-USA.xml#//city[name/text()=‘Washington D.C.’]”
           role=“capital”/>
           Washington D.C.
      </city>
      <city href=“file:cities-USA.xml#//city[name/text()=‘Albany’]”
           role=“state capital”/>
           Albany
      </city>
      <city href=“file:cities-USA.xml#//city[name/text()=‘New York’]”
           role=“city”/>
           New York
      </city>
      ⋮
   </cities>
   ⋮
</country>
```

Out-of-Line-Links. Out-of-line-links allow to create references not only inside documents, but also to create XML instances which consist *only* of links between other documents. This can be seen as a “personalized overlay” on arbitrary existing (autonomous) sources. Here, the link is not a directed link to a node, but a relationship (or a set of relationships) between a set of nodes; allowing also for multidirectional links.

Out-of-line-links consist of locator subelements which specify *potential* endpoints of links, and arc elements which specify the relationships between these endpoints:

```
<!ELEMENT linkelement (#PCDATA|locatorelement|arcelement)*>
<!ATTLIST linkelement as above >

<!ELEMENT locatorelement as above >
<!ATTLIST locatorelement as above
     type, href, id: ID , title, role >

<!ELEMENT arcelement (contentsmodel)>
<!ATTLIST arcelement
      xmlns:xlink   CDATA    #FIXED   “http://www.w3.org/xml/xlink/0.9”
      xlink:type      (simple|extended|locator|arc)  #FIXED “arc”
      xlink:from      IDREF  #REQUIRED
      xlink:to        IDREF  #REQUIRED
      xlink:show      (new|parsed|replace)  #IMPLIED
      xlink:actuate   (auto|user)  #IMPLIED  >
```

Example 3.21 (Out-of-line Links)
For the distributed MONDIAL *database, the memberships of countries in organizations are stored in mondial-memberships.xml:*

```
<!ELEMENT memberships (country*, organization*, membership*)>
<!ATTLIST memberships xmlns:xlink ...
                      xlink:type  (simple|extended|locator|arc) #FIXED "extended" >

<!ELEMENT country EMPTY>
<!ATTLIST country  xmlns:xlink ...
                   xlink:type  (simple|extended|locator|arc) #FIXED "locator"
                   xlink:href  CDATA #REQUIRED
                   id          ID    #REQUIRED >

<!ELEMENT organization EMPTY>
<!ATTLIST organization  xmlns:xlink ...
                        xlink:type  (simple|extended|locator|arc) #FIXED "locator"
                        xlink:href  CDATA #REQUIRED
                        id          ID    #REQUIRED >

<!ELEMENT membership EMPTY>
<!ATTLIST membership  xmlns:xlink ...
                      xlink:type  (simple|extended|locator|arc) #FIXED "arc"
                      xlink:from  IDREF #REQUIRED
                      xlink:to    IDREF #REQUIRED
                      membership_type CDATA #REQUIRED >
```

mondial-memberships.xml consists now of only one memberships *element:*

```
<memberships>
   <country id="D" xlink:href="…/countries.xml#id('D')" />
   <country id="F" xlink:href="…/countries.xml#id('F')" />
   <country id="I" xlink:href="…/countries.xml#id('I')" />
   <organization id="EU" xlink:href="…/organizations.xml#id('org-EU')" />
   <organization id="UN" xlink:href="…/organizations.xml#id('org-UN')" />
   <organization id="NATO" xlink:href="…/organizations.xml#id('org-NATO')" />
   <membership xlink:from="D" xlink:to="org-EU" membership_type="member"/>
   <membership xlink:from="D" xlink:to="org-UN" membership_type="member"/>
   <membership xlink:from="CH" xlink:to="org-UN" membership_type="observer"/>
</memberships>
```

The XPointer and XLink working drafts [XPt00,XLi00] specify how to *express* inter-document links in XML. There is not yet an official proposal how to *handle* XLinks in queries and applications. Section 13 describes general considerations on traversing XLinks.

3.9 XML Querying Data Model and Algebra

The recent XML Querying Data Model [XMQ01b] and XML Querying Algebra [XMQ01a] provide the formal background for XML querying by defining types and operators.

3.9.1 XML Querying Data Model

The XML Querying Data Model defines formally the information contained in the input to an XML Query processor in terms of trees using a *node-labeled* tree representation augmented with *node identity*. It refines and formalizes the basic XML data model (cf. Section 2.1.1) which also provided the base for the DOM API (cf. Section 2.1.3).

The basic concept in the data model is a *node*, which is of one of nine node types: *document*, *element*, *value*, *attribute*, *namespace*, *processing instruction*, *comment*, *information item* (an opaque value), or *node reference*. For every node, *accessors* are defined which return information on the node. The node types which are relevant for this work are described below:

Document: a document node is distinguished by a *URI reference*, and a non-empty forest of child nodes (exactly one of them must be an element node).
The *root* accessor returns the unique root element node of a document.

Element: an element node has a *tag* and consists of an unordered forest of namespace nodes, an unordered forest of attribute nodes, and an ordered forest of child nodes (containing nodes of the types element, value (for `PCDATA` children), processing instruction, comment, and information item). The namespace nodes give the namespaces which are declared on this element (note that an element may belong to several namespaces).
The accessors *name*, *namespaces*, *attributes*, *children*, *type*, and *nodes* return the element node's constituent parts (*nodes* returns an ordered forest containing the attribute nodes followed by the children). The *parent* accessor returns its (unique) parent node in the tree.

Attribute: attribute values have a name and a value. The value must be of any *simpleType* as defined in XML Schema (Datatypes).

Value nodes: value nodes contain the actual information, i.e., `PCDATA` contents. They can be of the primitive XML Schema datatypes.
The accessors *type*, *string*, *parent*, *localname*, and *namespace* return the corresponding information of value nodes.
The *referent* accessor returns an ordered forest of element nodes associated with an `IDREF` value.

Reference nodes: reference nodes are a mechanism to encapsulate the identity of nodes in a given instance. The accessor *deref* returns the node referred to by the reference node. Note that `IDREF`s are handled as values above.

In addition to nodes, the data model supports ordered (lists) and unordered (bags) forests of nodes which provide the formal base for node lists (e.g., *contexts* when evaluating XPath *locationSteps*, and (result) *nodeSets* of XPath expressions). As usual for lists and bags, the operators *empty*, *append*, *head*, *tail*, *add*, *empty*, *some* (chooses some element), and *union/diff/intersect* are defined.

3.9.2 Constraints of the Data Model

The data model imposes several constraints and requirements on the applications (mainly, querying) which use it. Most of these problems stated in [XMQ01b, Section 7] are too restrictive when dealing with data *manipulation* (cf. Section 8.5), and even more, when investigating data integration (see Section 11):

- The concept of *node references* (recall the problems with resolving `IDREFS` by string operators in XQL and XML-QL) is defined, but the actual technique of handling references for dereferencing is not implied.
- Node identity: two nodes are identical if they *are* the same node. This leads to problems when trying to handle nodes originating from several documents which have been identified to have the "same semantics".

- Unique parent (integration): this problem is closely related. Under this assumption, nodes originating from different documents may not be identified or fused.
- Unique parent (inserting/copying): if a new tree is to be generated from an old one, the elements cannot be "reused" directly (which would result in two parents) but have to be copied. Then, the maintenance of reference attributes gets difficult (see also Sections 3.12 and 8.5).

The above constraints are problematic for data integration, where corresponding elements in different sources have to be identified, merged, and a result tree consisting of information from the sources has to be generated. In the XML Query Data Model, this is only possible by creating the result tree from scratch without using elements from the sources. Since (deep-)copying of whole subtrees is expensive both wrt. computation time and (main memory) storage, this is not always a favorable solution. Additionally, the maintenance of references is problematic. For data integration in XPathLog/LoPiX, we use a different data model which allows for *overlapping* trees, elements having multiple parents, and element fusion, and synonyms for names (see Section 11).

3.9.3 XML Querying Algebra

The *XML Querying Algebra types* describe the structure of classes of XML instances and elements – similar to a DTD or an XML Schema specification. Since every query implicitly defines a class of XML instances – the instances which are generated when evaluating the query wrt. a database, also a query has a type. An *operator* is an instruction how to compute a result from given operands. From the algebraic point of view, a query is an *operator-tree* which specifies how the query is built from elementary operators. Based on the operator tree, the algebra defines the *semantics* of queries by specifying both the result of evaluating queries wrt. XML instances, and the *result types* of queries:

- "Static Semantics": the result types are derived by structural induction according to the given *type-inference* rules, and
- "Dynamic Semantics": the results themselves when evaluating a query wrt. an XML instance is defined by *value-inference* rules.

The datatypes are – naturally – closely related to the *complexType* concept of XML Schema. While the data model and the algebra serve for the formal background of the types and the operators, XML Schema serves for expressing the type system in XML. The algebra takes the atomic datatypes from XML Schema and the algebra's *external* type system, i.e., the type definitions associated with results, are *complexTypes* according to the XML Schema definition. The internal types are more expressive than those of XML Schema.

Complex datatypes are defined as nested regular expressions, giving the names of properties (i.e., attributes and subelements) together with their cardinalities. Also, the allowed order of subelements is specified (",": sequence; "&" both, in arbitrary order; |: choice; $t\{a,b\}$: repetition of a-and-then-b for t times).

The algebra does not yet support `IDREF(S)` [XMQ01a, App.B, Issue 0007]. There are several other open issues, such as XPath axes, a three-valued logic for null values, regular path expressions, global vs. local types etc.

Example 3.22 (XML Query Algebra: Datatypes)
The mondial *and* country *types of the* MONDIAL *database are specified as follows (according to the XML Schema given in Appendix B):*

```
type Mondial = mondial [country {1,*} &
                        continent {1,*} &
                        ...  ]
```

```
type Country = country [@car_code [ IDREFS ] {1,1} &
                        @capital [ IDREF ] {1,1} &
                        @memberships [ IDREFS ] {0,1} &
                        @area [ Number ] {1,1},
                        name [ String ] {1,1},
                        population [ Integer ] {1,1},
                        indep_date [ Date ] {1,1},
                            ⋮
                        ethnicgroups [ CulturalInfo ] {0,*},
                        religions [ CulturalInfo ] {0,*},
                        languages [ CulturalInfo ] {0,*},
                        border [ Border ],
                        ( province [ Province ] {1,*} |
                            city [ City ] {1,*})
       ]
type CulturalInfo = culturalinfo [mixed @percentage [Decimal] {1,1}]
type Border = border [@length [Decimal] {1,1} &
                      @country [ IDREF ] {1,1}
       ]
```

Instances have also a representation in the algebra syntax:

```
mondial = mondial
    [ continent [ name ["Europe"] ,
                  area ["9460138"]
                ]
      ⋮
      country [ @car_code ["D"],
                @capital ["city-berlin"],
                @memberships ["org-un org-eu ..."],
                @area ["356910"],
                name ["Germany"],
                population ["83536115"],
                indep_date ["23 05 1949"],
                    ⋮
                ethnicgroups [ @percentage ["95.1"],
                               "German"
                             ]
                ethnicgroups [ @percentage ["2.3"],
                               "Turkish"
                             ]
                ⋮
              ]
      ⋮
```

The algebra associates a result type (or, will do so in its final version) with every query. Queries in the algebra are *expressions* over operators, constants and names. In contrast to the relational algebra which consists of *selection*, *projection*, *cartesian product/join*, *union*, and *minus* with derived standard operators *intersection* and *division*, the XML Query algebra provides a large set of operators as described below (for details see [XMQ01a, Figure 2]).

1. projection + selection: NameExp[Exp] and @NameExp[Exp] select elements or attributes by name (or selection expressions over names, wildcards, etc.),
2. sequence/union, difference, intersection,
3. arithmetic, boolean, and equality operators, function applications,
4. iteration: for *var* in *expr* do *expr* for iteration. There is no join operator, it has to be specified as nested-loop-join using two for iterations. Note that the for-joins to not commute, since the order of elements in the result is relevant.
5. conditional: if *expr* then *expr* else *expr*,
6. local binding: let *var*=*expr* do *expr*,
7. sorting: sort v in *expr* by *expr*,
8. match: match *expr* *caseRules*, where *caseRules* is case *var* : *Type* do *exp* *caseRules*.

Here, (1)–(4) implement the operations known from the relational algebra, whereas the remaining constructs deal with the additional complexity due to the nested structure and order.

So far, the core algebra is quite far from the concepts known from XPath. The XPath "operators" are defined as *reducible* algebra expressions – here, [XMQ01a, Appendix A] shows how these XPath expressions are translated into core algebra expressions.

Example 3.23 (XML Query Algebra)
The XPath expression mondial/country is equivalent to

```
for n in nodes(mondial) do
    match n
        case c : country[AnyComplexType] do c
        else ()
```

which means: iterate over all nodes in mondial. For every node n, if c is the result from matching n with "country[AnyComplexType]" (i.e., a projection on the element's tag if this is "country", and all its contents), then return c (which is in fact the complete country element if n is a country element). If c is nothing (i.e., n does not match "country[AnyComplexType]" which happens exactly if it is of an element type other than country*, return the empty sequence.*

Thus, the core algebra is a low-level specification of operators – corresponding to an implementation-level specification of SQL expressions – which are much more powerful than the XPath operators. On an intermediate level, [XMQ01a, Appendix A] also introduces a where *cond* do *expr* construct.

A potential application of the algebra for query rewriting and optimization evolves from the *Quilt* language [CRF00, RCF00] (which has then been turned into the XQuery working draft [XQu01]) which is described in the following section.

In the XML Query Algebra Working Draft, an *unordered* model is explicitly distinguished [XMQ01a, Section 2.8], providing e.g., a symmetric join where traditional optimizations are applicable.

3.10 Quilt

In some sense incorporating an "evolution step", Quilt [CRF00,RCF00] is the first query language that *embeds* XPath syntax into higher-level constructs (similar to SQL/OQL [ASL89]) (as has already been done in XML Schema and XLink for other tasks). Quilt queries consist of a series of clauses that declaratively describe

- what information is to be used,
- which additional conditions apply, and
- how the result is to be constructed.

The keywords in Quilt clauses are `FOR - LET - WHERE - RETURN` (FLWR; pronounced "flower"):

```
FOR variable IN xpath-expr
LET additional_variable := xpath-expr
(FOR | LET)*
WHERE filters
RETURN xml-expr
```

The `FOR` clause defines variables which are bound to the elements of iterating over the result set of XPath expressions. Additional variables may be defined in the `LET` clause, computed from the ones defined in the `FOR` clause. The variables in the `FOR` clause iterate over the corresponding *xpath-expr*, whereas the variables in the `LET` clause are bound to the *result* of the corresponding *xpath-expr*, i.e., the may be bound to a *node set*. Variables defined in `FOR` or `LET` clauses can then be used in subsequent `IN` clauses. The result from the `FOR` and `LET` clauses is a sequence of variable bindings. The `WHERE` clause – similar to SQL/OQL – states additional conditions which tuples of variable bindings are used for generating the result, using the XPath filter syntax, augmented with some syntactic sugar. Then, the `RETURN` clause generates an XML subtree for each variable binding.

Example 3.24 (Quilt)
The following Quilt query returns a result document which consists of all countries which have an area of more than 1000000 km^2 and contain more than 10 cities with all their cities as subelements:

```
FOR $c IN document("mondial.xml")//country
LET $cities = $c/city
WHERE $c/@area > 1000000 and count($cities) > 10
RETURN
      <bigcountry area = $c/@area>
         <name>$c/@name</name>
         $cities
      </bigcountry>
```

Note that for every value of $c, $cities is bound to the nodeset containing all city subelements of $c.

As the name says – a *quilt* is a patchwork carpet – Quilt is a patchwork, influenced from many querying languages:

- SQL/OQL: queries consist of a series of clauses.
- OQL: a functional language whose expressions which may be arbitrarily nested.
- XPath/XQL/XSL Patterns: navigation in XML trees is done by XPath expressions. Quilt *extends* XPath with a derefencing operator for resolving `IDREF(S)` attributes, e.g.,

//country[@car_code = "D"]/@capital→/name/text()

selects the *capital* reference attribute from the *country* element whose *car_code* attribute has the value "D", dereferences it (yielding a *city* element node) and returns the text contents of the *name* subelement.

Additionally, the `BEFORE`, `AFTER`, `INTERSECT`, and `EXCEPT` constructs from XQL are supported.

- XQL: a `FILTER` construct is provided for output projection which shows some similarities with XQLs output and grouping operators.
- XML-QL:
  - the structure of Quilt queries as a whole is very similar to XML-QL (`WHERE` *xml-pattern* `IN` *url* `CONSTRUCT` *xml-pattern*). Similar to XML-QL (and SQL), Quilt does not support recursive queries.
  - information is propagated from the extraction (`FOR - LET - WHERE`) part to the construction (`RETURN`) part via *variable bindings*,
  - where in both cases, the result is generated by instantiating an XML pattern with variable bindings (possibly with nested FLWR expressions which contribute subtrees for grouping).
  - The main difference is that the extraction part in XML-QL also uses an XML pattern which is *matched* whereas Quilt uses iterators and collections over XPath expressions.

Example 3.25 (Quilt: Dereferencing)
Consider the query given in Example 3.4:

id(//organization[id(./@seat) = id(id(./member/@country)/@capital)]/@seat)/@name

The equivalent expression in Quilt is

//organization[@seat→= members/@country→/@capital]/@seat→/name/text() .

The Quilt language provides the "usual" operators used in database queries:

Joins. Quilt allows for *joins* in the `FOR` clause by specifying several "*var* `IN` *xpath-expr*" arguments, or by a sequence of `FOR - LET` clauses. Each `FOR - LET` clause may contain references to variables defined before, allowing for "correlated joins" such that the domain of the second join already depends on the current value of the first one (which is not possible in SQL, but is allowed also in OQL).

Additional *join* functionality is provided by using FLWR expressions in the `RETURN` part. Here, also the inner FLWR expression may access variables from the outer clause.

Using nested FLWR expressions in the `WHERE` clause provides only restricted join functionality for evaluating conditions since the values from the inner subquery cannot be communicated to the result (similar to SQL/OQL).

Selection. Selection functionality is explicitly provided by the `WHERE` clause, but also the XPath expressions (filters, existential semantics) provide functionality which is implemented in SQL by selection.

Projection. Projection is supported by the definition of variables in the `FOR - LET` clause, and mainly by a `FILTER` operator which extends the XPath syntax: the expression

xpath-expr$_1$ FILTER *xpath-expr*$_2$

results in a tree which contains exactly the nodes of the result set of $\mathsf{xpath\text{-}expr}_1$ which are selected by $\mathsf{xpath\text{-}expr}_2$, retaining the document structure and order (similar to XQL). Nodes are taken *without* attributes or subelements, i.e., only the tags are kept. The attribute nodes and element nodes which should occur in the result have to be selected explicitly. If an intermediate node n in a tree does not qualify, but some of its children qualify, those occur at n's position.

Example 3.26 (Quilt: Filter)
The expression

//mondial FILTER //country | //country//city

produces only a forest of the form

<country> <city/> </country>
<country> <city/> <city/> <city/> <city/> </country>
<country> <city/> <city/> <city/> </country>
⋮

(the mondial node does not qualify, and also no attribute nodes and no text nodes are projected). Thus, to get the intended result containing also the names, the filter condition must be extended:

//mondial FILTER //country | //country/@car_code | //country/name/text()
| //country//city | //city/name/text()

which then generates

<country car_code="A"> Austria <city>Vienna</city> </country>
<country car_code="D"> Germany <city>Berlin</city><city>Hamburg</city>
<city>Munich</city> </country>
⋮

Note that the `FILTER` operator changes the tree structure: elements which have been descendants before may become children when intermediate elements are omitted in the projection. E.g., in the above example, the intermediate *province* elements do not occur in the result, instead, the *cities* are directly children of the *countries*.

Restructuring. There is additional restructuring functionality which provides a combination of selection and projection:

- the XPath expressions in the `FOR` clause are a combination of selection and projection (projection of the tree on some of its nodes/subtrees),
- the restructuring when generating the result in the `RETURN` clause where XML subtrees are constructed is much more expressive as the "result generation" by projection only in the `SELECT` clause in SQL. The use of variables at tag position is allowed in the `RETURN` clause.

Note that also the `SELECT` clause in SQL (especially, in the SQL-3 standard with object-relational extensions) does not only provide projection, but also restructuring and generating functionality.

Additional Operators. Several operators are allowed which have also been present in SQL/OQL: The `FOR` clause may be augmented with a `DISTINCT` specification. A `SORTBY` *expr* clause is allowed after the `RETURN` clause for sorting the nodes produced by the `RETURN` clause. The SQL/OQL `ANY` and `ALL` operators have a more explicit syntax in Quilt, using a

WHERE SOME/EVERY *var* IN *expr* SATISFIES *predicate*

clause. Aggregate functions similar to SQL/OQL are also provided.

Kweelt. KWEELT [Sah00] is an extensible, open-source Quilt implementation in Java based on a DOM implementation. Here, the dereferencing operator is further extended by adding *dereferencing hints* which specify where to search for the target of a reference: a dereferencing step is of the form

$$\mathit{refAttr}\rightarrow\{\mathit{elementtype}_1@\mathit{attr}_1,\ \ldots,\ \mathit{elementtype}_n@\mathit{attr}_n\}\ .$$

The hint then means that the target of the reference is of one of the types $\mathit{elementtype}_1, \ldots,$ $\mathit{elementtype}_n$ where the attributes $\mathit{attr}_1, \ldots, \mathit{attr}_n$, respectively, must be checked. The above selection expression translates as

//organization[@seat→{city@id} = members/@country→{country@car_code}/@capital]
/@seat→{city@id}/name/text() .

Practical experiences with KWEELT showed that the system is still very slow.

Issues. An important aspect when implementing and using Quilt is that *selection* functionality is provided both by the XPath expressions in the `FOR` clause, and in the `WHERE` clause. In some sense the problem corresponds to the use of `SELECT` statements in SQL in the `FROM` clause for restricting the set of tuples participating in the join, or to apply selection in the `WHERE` clause only.

In SQL implementations, an internal (theoretically, algebra-based) query optimizer rewrites the statements into an optimal operator tree. Regarding Quilt, this is obviously an aspect where the XML Query Algebra (see Section 3.9.3) should be useful.

Similar to XML-QL, when (deep)-copying elements into the result tree, the result tree may contain dangling reference attributes.

3.11 XQuery

With some minor revisions, Quilt (using the XPath 2.0 semantics now) has become the *XQuery* Language [XQu01] (Feb. 2001 Working Draft). Since most design issues of the language have been solved and described above for Quilt, mainly the "W3C-specific" issues of integrating Quilt with the recent developments for the *XML Query Data Model and Algebra* remained.

Quilt introduced a dereferencing operator "→" for reference attributes (which has already been extended by the KWEELT implementation). In XQuery, the operator uses a *name test* that specifies the expected name of the target element (the wildcard "*" as nametest accepts all types of target nodes):

//country[@car_code = "D"]/@capital→city/name/text() .

If several documents should be integrated, *named documents* are allowed (cf. the notion of *document nodes* in the XML Query Data Model). Using the function document(*string*) (cf. the document(...) extension function in XSLT), the document node representing the document named *string* can be accessed. Document nodes may serve as starting point for XPath expressions, e.g.,

document(mondial)/country[@car_code = "D"] .

For data integration, the documents to be integrated in general use their own namespaces. XQuery allows for accessing namespace definitions and assigning them to constants which then can be used for selecting navigation steps according to the namespaces:

NAMESPACE mondial = "http://.../mondial/names"
NAMESPACE cia = "http:cia.gov//factbook/names"

Then, the query

document("mondial.xml")//mondial:country/cia:*/name()

navigates in the document *mondial.xml* to the country elements in the *mondial* document, then navigates to all elements whose name is contained in the *cia* namespace and outputs their names (i.e., outputs all names of subelements of *mondial* countries which are also names in the *cia* namespace).

In XPathLog/LoPiX, we use a different approach for handling multiple documents with (possibly overlapping) namespaces (see Section 11).

Quilt has been designed before the XML Query Algebra has been defined, using the type system of XML Schema which has minor differences wrt. the one defined in the XML Query Algebra. A preliminary mapping from XQuery expressions to the Algebra is given in [XQu01, Appendix E]. The type systems of XML Schema and Quilt/XQuery will be aligned with the "reference" type system of the XML Query Algebra.

Comparing XQuery with the XML Query Requirements (cf. Section 3.6) the situation is as follows:

- XQuery covers the aspects of both *document-oriented* and *data-oriented* documents,
- it is declarative, not enforcing a particular evaluation strategy,
- it supports the XML 1.0 datatypes (and the primitive datatypes of XML Schema). The *complexTypes* of XML Schema have to be refined wrt. the XML Query Algebra,
- XQuery partially supports references within a document (dereferencing operator for `IDREF(S)`). References from one XML instance to another (XLink) are not supported,
- currently it has only one language binding, the FLWR expressions. A syntax which is expressed in XML is still missing. Nevertheless, an XQuery version which embeds XPath into XML-syntax XQuery FLWR expressions (similar as XSLT-style combines XPath and its own elements into XML syntax) is just syntactic sugar.

3.12 Updating XML

The languages presented above (representing the state of the art in early 2001) do not (yet) provide constructs for *updating* XML data. Applications can change existing XML documents based on the DOM model, but there is no *XML update language*. Excelon [eXc] provides a proprietary extension to XSLT, called *XUL (XML update language)* (cf. Section 3.13).

[TIHW01] describes a proposal for updates in XML, based on the XML Query Data Model [XMQ01b]. A language-independent definition of updates is given; the only assumption is that the updates are executed in an environment based on variable bindings (as, e.g., in XML-QL or Quilt/XQuery). The *target* of an update is the element node which is updated. The following operations can be applied to *target* – provided with suitable bindings of other variables:

- Delete(*child*): if *child* is a member (i.e., a subelement (including `PCDATA`), an attribute, or a reference in an `IDREF(S)` attribute), it is removed:

  ```
  $target = //country[@car_code="B"]
  $prov = //country[@car_code="B"]//province[name="Brabant"]
  $target.DELETE($prov)
  ```

  deletes the province subelement of Belgium which represents the province of Brabant. Note that the whole subtree representing Brabant is deleted; including the city element which represents Brussels. Thus, there will be *dangling references* (e.g., *belgium*/@capital and *eu*/@seat). As all the above languages allow for dangling references in their output trees, [TIHW01] also allows them explicitly.

  Note that for `IDREFS`, it is also possible to remove single references, e.g., the membership of Belgium in the EU; in this case, XPath does not provide functionality to address the item to be deleted.

- Rename(*child,name*): if *child* is a non-PCDATA member (i.e., a subelement or an attribute), it is given a new name. For IDREFS, only the whole list can be renamed, the renaming of single items is not possible. Here, [TIHW01] does not specify what happens if an attribute is renamed to a name which already exists.
- Insert(*content*): *content* can be PCDATA, an element, an attribute or a reference. For details see below – insertion is critical due to the unique-parent-requirement in the XML Query Data Model.
- InsertBefore(*ref,content*): *ref* and *content* must either both be element or PCDATA, or both references. Then, *content* is inserted before *ref*. Analogously for InsertAfter(*ref,content*).
- Replace(*child,content*) is an *atomic* replace operation, equivalent to InsertBefore(*child,content*) followed by Delete(*child*).
- SubUpdate(*patternMatch*, predicates, updateOp): starting with *target*, all elements addressed by *patternMatch* are selected, these are then filtered wrt. *predicates*. For the remaining elements, *updateOp* is applied. Note that *subUpdates* can often be replaced by suitable constructs of the surrounding language (e.g., FLWUpdate expressions in XQuery+updates).

Based on the abstract operations, an extension of XQuery with `FOR - LET - WHERE - UPDATE` clauses is given where the `UPDATE` clause is of one of the forms:

DELETE $*child*
RENAME $*child* TO *name*
INSERT *content* [BEFORE|AFTER $*child*]
REPLACE $*child* WITH *content*
FOR $*var* IN *xpath-expr* WHERE *preds update*

where *content* can either be a variable, or an XML pattern containing variables for generating new structures.

Insertions. Since [TIHW01] uses the XML Query Data Model [XMQ01b], the constraints described in Section 3.9.2 apply:

- Unique Parent (inserting/copying): if a new tree is to be generated from an old one, the elements cannot be "reused" directly (which would result in two parents) but have to be copied.

That means, if any variable in *contents* is bound to a subtree, this subtree must be copied when inserting it (or replacing another element by it), leading to the problems described in Section 8.5 for generating new IDs and adapting IDREFs.

3.13 Excelon and XUL

Excelon [eXc] is a commercial XML-based B2B platform. As already stated in the introduction, the first versions implemented XQL and XSLT for querying and manipulation. With the evolving XPath standard from XQL, XSL Patterns, and XPointer, since version 3.0, all XQL-only features have been removed and replaced by corresponding XPath features. The main goal of the eXcelon platform is to act as an XML database in a B2B-environment by providing Java interfaces for database access using standard DOM and XPath APIs.

An interesting extension by eXcelon is *XUL (XML Update Language)* which extends XSLT by suitable constructs in the xln: namespace. "Updategrams" are XML instances which consist of an <xlnupdate version="1.0"> element whose contents is a sequence of <foreach>, <update>, and <remove> elements which specify updates to the database:

- A <foreach select="*xpath-expr*"> element applies its contents to all nodes selected by *xpath-expr*. Its contents consists of <foreach>, <update>, and <remove> elements.
- An <update select="*xpath-expr*"> element adds attributes or elements to the node selected by *xpath-expr*. The contents consists of XSLT <element> and <attribute> elements which describe the updates.
- A <remove select="*xpath-expr*"> element removes the element(s) selected by its *xpath-expr*.

Example 3.27 (XUL)
The following "updategram" deletes all cities which are capitals and changes the capital *reference attribute to* Timbuktu. *Note that also all* organization/@seat *references to deleted capitals have to be removed.*

```
<xlnupdate version="1.0">
  <remove select="//organization[@seat=//country/@capital]/@seat"/>
  <foreach select="//country">
    <remove select="id(@capital)"/>
    <remove select="@capital"/>
    <update select="."/>
      <xsl:attribute name="capital">
        <xsl:copy-of select="//city[name='Timbuktu']/@id"/>
      </xsl:attribute>
    </update>
  </foreach>
</xlnupdate>
```

3.14 YAXQL

YAXQL [Moe00] is an XML querying language which *is* in XML syntax. Similar to XSLT, it defines querying language elements which contain XPath expressions as attribute values and element contents. The result is also created by YAXQL command elements. Similar to XML-QL and Quilt/XQuery, the matching part of a query generates variable bindings which are communicated to the constructing part.

Queries in YAXQL consist of four parts with the responsibilities to

- produce variable bindings,
- restrict the collection of produced variable bindings,
- project on relevant variables, and
- construct a result.

A complete query with result construction is an <xql:query-construct> element which has a name attribute, which associates a name with the query for identifiying it in case of reuse. The <xql:query-construct> element contains an <xql:query> element which represents the query, and an <xql:construct> element which constructs the result. Variable bindings are communicated from the former to the latter. Queries may be parameterized.

Inside the <xql:query> element,

- the free (input) variables of the query are declared by <xql:declare var="*var*" ...> elements,
- additional variables are assigned with values using <xql:match var="*var*" select="*xpath-expr*"> elements. <xql:match> elements can be nested,

- variables can also be bound by <xql:iterate> elements which specify iteration, e.g., over a whitespace-separated list of items in a string (tailored for the handling of `IDREFS` attributes),
- a query may contain subqueries, either as <xql:query> elements or by reusing another query by an <xql:query-reference href="*query-name*"> element which contains <xql:bind> elements to bind the results from the referenced query to variables of the surrounding query,
- <xql:project> elements serve for binding variables to nodes, text, or even XPointers, using <xql:bind var="*var*" select="*xpath-expr*"> elements.

Recursive queries are defined by <xql:recursive-query name="*name*"> elements, containing <xql:union> elements which in turn contain one or more <xql:query elements> which may now refer to the surrounding <xql:recursive-query> element.

The <xql:construct> element then constructs a result similar to XSLT, using the variables bound in the <xql:query> part. Here, additional elements an be used for specifying iteration, concatenation etc.

There is no reported implementation of YAXQL.

3.15 XML Metadata: DTDs, XML Schema, and XML Query Algebra

In practice, since the DTDs are the "original" way to describe structural information on XML documents, most XML tools support DTDs, and thus, most XML instances come with DTDs.

An important argument for DTDs is that standard validating parsers use them; although also XML Schema-aware parsers/validators have recently been presented (see `http://www.w3.org/XML/Schema`). On the other hand, DTDs give only incomplete information on what usually is understood as schema information in databases: especially, the support for cardinalities is only restricted.

With usual XML querying languages, there is only implicit access to the DTD - it does not fit into the XML data model (commercial tools such as Tamino [Sof] and eXcelon [eXc] provide access to the schema description of stored documents). Nevertheless, as stated above this implicit access is crucial for validating parsers, and for parsing documents unambiguously (e.g., distinguishing `ID` attributes from `CDATA` ones). In contrast, for querying, as shown in Section 3.2.2, it is not necessary to distinguish `IDREFS` from `NMTOKENS` – everything can be dereferenced.

In contrast, XML Schema documents are valid XML documents according to the XML Schema DTD which specifies the structure of XML Schema instances. Thus, applications which are aware of the special semantics of these tags can access them using standard XML querying languages such as XML-QL, or Quilt/XQuery, similar to the *Data Dictionary* in relational database systems.

While both DTDs and XML Schema are designed for practical use, the XML Query Algebra defines a type system for formal use. Comparing the underlying metadata models, DTDs, XML Schema, and the XML Querying Algebra define different notions of typing:

- DTD: Every element (type) defines a "type" whose properties are defined by the contents model of the <!ELEMENT> declaration and by the <!ATTLIST> declaration. The DTD typing is a "1-level-typing".

  One of its main drawbacks for practical use is that it does not distinguish between element types and subelement names: in the MONDIAL example, both *countries* and *cities* have population subelements, but with a different structure. This is not expressible with a DTD.

- XML Schema: element types are defined by *complexType* elements. The declarations of subelements and attributes may either refer to existing element and attribute definitions (1-level), or define local types (nested-types).

- XML Query Algebra: A type consists of the description of the complete nested structure without referring to other (non-basic) types (at least in the current version).

For practical aspects, one-level-typing models are better to handle (the metadata extension of XPathLog (see Section 10) is also a 1-level typing model).

3.16 Discussion

The development of languages in the XML world is now centered around the XPath concept which itself developed from XSL Patterns, XQL, and XPointer. Based on the "stable" notion of XPath, the "second-level" concepts of XSLT, XMLSchema, XLink, XQuery etc. are defined which combine XML syntax with XPath,

- defining special XML elements in reserved namespaces which represent constructs of the respective (querying, updating, linking, etc.) language,
- embedding XPath expressions as attributes and element contents for dealing with XML instances.

Currently, XQuery represents the W3C state of the art in XML querying. The next extensions are foreseeable:

- XQuery does not yet – as required in the XML Query Requirements [XMQ01c] – provide a syntax which is completely in XML. Nevertheless, an XQuery version which embeds XPath into XML-syntax XQuery FLWR expressions is just syntactic sugar – just a question of time.
- Compared with SQL, XQuery is a pure querying and transformation language, it does not yet support updates. [TIHW01] describes a proposal for updates in XML, see Section 3.12.

At the same time, the non-W3C language XML-QL has also become an established framework, implemented in [DFF+99c], and used in several projects (e.g., SilkRoute [FTS00], or MIX [BGL+99]; see also Section 16.1).
With *YAXQL* [Moe00], an XML querying language in XML syntax has been presented; also a data manipulation extension of XSLT has been defined by Excelon in *XUL (XML Update Language)* [eXc].

Dangling References in the Result Tree. All the above querying languages share the problem of handling dangling references in the result tree: Generally, for *every* reference attribute which is added to the result tree (often, by deep-copying elements bound to variables), it has to be checked if the target of the reference(s) also belongs to the result tree. Considering the fact that the splitting of `IDREFS` is not satisfying by the above languages, this check is in general not possible to be integrated into the query.

Early Predecessors. The above sections did not describe earlier predecessors of XML, handling semistructured data, such as TSIMMIS/MSL/OEM/Lorel [GMPQ+97, AQM+97a] (starting from integration of heterogeneous data), STRUDEL/STRUQL [FFK+98] (from the point of view of a Web site management system; STRUDEL/STRUQL served as a base for XML-QL), or F-Logic/FLORID [HKL+98, KLW95] (originally an object-oriented logic-based framework). They will be compared in Sections 4.2 and 16.1.

Preview. In the subsequent sections, *XPathLog*, a logic-programming language based on XPath is defined, extending XPath in Datalog style: the rule body serves for generating variable bindings which are used in the head for *extending* the current XML database. In contrast to the languages presented above, XPathLog does not only produce a standalone result tree which is isolated from the database, but implements database manipulation directly. In contrast to, e.g., XSLT, the language does not use additional constructs whose semantics has still to be defined: the only semantics prerequisite is the bottom-up evaluation strategy of Datalog or any other rule-based language. Thus, the implementation in LoPiX (see Section 15) could be based on the evaluation component (and even the storage component) of the Florid [HKL+98, LHL+98] system, an implementation of F-Logic [KL89, KLW95].

4 DATABASE VS. DOCUMENT POINT OF VIEW

The XML data model – and semistructured data in general – applies both to *documents* and to *databases*: The SGML language was originally defined for *documents* in the publishing area. On the other hand, the interest in research on *semistructured* data in the late 1990s was motivated from the *database* community, searching for a data model for *data integration* and a data format for *electronic data interchange*. With the latter, also the combination of document-oriented aspects with database aspects in a single instance was an important motivation to go beyond classical data models.

4.1 Analysis

Data Model. The XML data model is a hierarchical model which defines an ordered tree with attributes that can easily be interpreted as a document. The natural relationships in documents are either (i) substructures, or (ii) references to other parts of the document (where the term *reference* here means simply a cross-reference in a document). In XML, the nested elements define a document structure whose leaves are the text contents. Elements (i.e., the structuring components) are annotated by attributes which do *not* belong to the document contents. Inside the tree, cross-references (`IDREF` attributes) are allowed.

From the document point of view, the DTD (inherited from the document-oriented SGML area) is suitable as a specification of the document structure (which cannot really be called a "schema"). The fact that the DTDs do not allow for specifying the target element type for `IDREF` attributes indicates the minor importance of references in this aspect.

In contrast, for databases, a hierarchical structure is in general not intuitive. Here, several kinds of relationships have to be represented, between substructures and pure references. When using XML for a database-like application, these relationships have to be represented by reference attributes. On the other hand, order is often not relevant in databases.

For the database point of view, DTDs are insufficient: already expressing that an element must contain some subelements in *any order* is longwinded (cf. Section 2.5 – the task becomes even more complicated when cardinalities are given). Additionally, there is no means for specifying the target element types of reference attributes – considering the fact that relationships must in general be represented by reference attributes, this is a serious weakness.

With XML Schema, there is a metadata description formalism which is better tailored to the needs of the database community.

Languages. The development of the navigation and querying languages (XQL, XSL Patterns, XPath) also reflects the origin in the document community: navigation along references requires explicit constructs (via semijoins and the id function). The design of XSLT as a *transformation* language based on patterns which are recursively applied also shows its origin in the document area, traversing a tree. Using XSLT as a database language requires completely different thinking as e.g., in SQL/OQL (or more general, the notion of a "transformation language" is unusual in the database area). An update language *XUL* based on XSLT has been defined by Excelon [eXc], nevertheless, the basic idea is the recursive application of functional-style patterns as a scripting and transformation language.

The more recent languages XML-QL and Quilt/XQuery are strongly influenced from the database area. Their queries are declarative in the SQL/OQL style, containing subqueries and operators; the underlying theory is also based on operators related to the relational model (adapted to the nested structure). They handle references (which means, navigation along relationships in the database context) as "full" members of the data model as known from the object-oriented ODMG model.

The requirement that operations must be order-preserving effects the algebraic theory, e.g., the join is not commutative. Thus, it constrains the optimization strategies and effects also the implementations and their efficiency. Note that most languages are based on XPath which does *not* impose any constraints on the ordering of elements in its result sets (although most of the implementations are order-preserving) – thus, implementations rely on a global ordering of elements (see also the comparison in Section 8.2).

Data Integration. Data integration means *completely* different things in the document area and in the database area.

"Integration" of documents. Integration of documents is either a simple process, or it needs much more than only programmable restructuring:

When a large document is generated from *disjoint parts* (e.g., compiling lecture notes from a set of individual lectures), the task consists mainly of generating a large tree from smaller ones without actually touching the contents of the original trees.

On the other hand, when a document has to be generated from a set of documents which are not disjoint, but each of them describes the same general subject from different points of view, it is expected to be a manual process (e.g., compiling a thesis from several papers): each section must be generated by combining the contents of appropriate sections in the original documents. In more detail, paragraphs must be adapted and rewritten from the original documents in a copy&paste (&edit) manner.

The first task is completely covered by XSLT, whereas the second one is optimally supported by comfortable editors. In neither case, a "data integration language" in the sense of database languages is required.

Integration of database contents. Research on data integration in the database community has pointed out the need for powerful languages which allow for combining database contents with database metadata (i.e., schema information), e.g., SchemaSQL [LSS96b], Tsimmis/OEM/Lorel [GMPQ$^+$97], Strudel/StruQL [FFK$^+$98], or F-Logic [HKL$^+$98].

In contrast to documents, the source is seen as a database where objects and their properties are described; references are regarded as object-valued properties. Thus, there is not a dominating hierarchical structure, but every source defines a densely connected graph. The same contents can be represented by totally different XML trees.

Thus, a tree-oriented proceeding as in XSLT is not suitable. A language for XML data integration must support arbitrary navigation in the XML instance, preferably also starting at arbitrary entry points associated to constants. Then, the integration process is a complex task which cannot be done in a one-pass way (as with XSLT), but which must iteratively collect and refine a result view on the source graphs (cf. Sections 11 and 12).

On the other hand, in this context, the order of properties and children is in general irrelevant. In case the order is relevant, this can e.g., be expressed by attributes. When an "object" collects properties and relationships from different sources, there is no inherent order. As stated above, without order, the algebraic theory of a language is much easier, allowing for more efficient optimization strategies.

In the following, the language *XPathLog* is developed as an XML querying and data manipulation language. Although the main goal of the language is data integration, the language concept itself includes the handling of order. Later, when focussing on data integration in Sections 11 and 12, the order is ignored.

4.2 Solutions in Related Approaches

The pre-XML projects on semistructured data in the database area used unordered data models strongly influenced by the object-oriented paradigm, e.g., TSIMMIS/OEM/LOREL [GMPQ+97, AQM+97a] , STRUDEL/STRUQL [FFK+98], or F-Logic/FLORID [HKL+98,KLW95]. These projects in turn served as a base for projects in the XML area: XML-QL has been developed based on the experiences with STRUDEL/STRUQL (translating XML-QL queries into STRUQL). The XML-QL language is also used in the MIX (*Mediation in XML*) project [BGL+99] which has also been influenced by the Tsimmis architecture. The approach presented in this work, XPathLog and LOPiX, are based on F-Logic/FLORID. Thus, these projects are originally based on unordered, graph-based models.

XML-QL provides both an unordered and an ordered data model as described in Section 3.5. The unordered one is practically an object-oriented model: an edge-labeled graph whose nodes (equipped with *object ids*) are the elements, the leaves are the text elements, and attributes are associated with element nodes. The ordered model extends this model with a global order on nodes. MIX uses the XML-QL language, thus it is based on the same model.

The XPathLog/LOPiX project described in the present work is also based on an originally unordered graph model. In contrast to the XML-QL data model, the ordered variant enumerates the children of a node directly, not imposing a global order (see Section 5.3). As a consequence, it is e.g., allowed that e_1 is the first child of a node x, and e_2 is the second child, whereas for another node y, e_2 is the first child and e_1 is the second child. This may be useful when the contents of a source has to be reordered.

In contrast, the YAT/YATL (*Yet Another Tree Model/Language*) [CDSS99] came up already with an ordered tree model motivated by SGML (e.g., using DTDs). In [CCS00], the YAT system is turned into an XML system for data integration.

In the XML Query Algebra Working Draft, an *unordered* model is explicitly distinguished [XMQ01a, Section 2.8], providing e.g., a symmetric join where traditional optimizations are applicable.

5 XPATH-LOGIC: THE FORMAL FRAMEWORK

5.1 XPathLog: Adding Variable Bindings to XPath

In this section, XPathLog is presented from an intuitive point of view as a logic-based XML querying language which extends XPath [XPa99] (see Section 3.2) with Datalog-style variable bindings. It is the Horn fragment of XPath-Logic which is formally defined in Sections 5.4 and 5.5, after defining suitable logical structures in Sections 5.2 and 5.3. XPathLog will be defined formally in Sections 5.4 and 6.

Example 5.1 (XPath, Result Sets)
Consider the following excerpt of the MONDIAL *database [Mon] for illustrations. The complete DTD can also be found in Appendix A, a representation of this excerpt as a graph can be found in Figure 5.1.*

```
<!ELEMENT mondial (country+, organization+, ...)>
<!ELEMENT country (name, population, city+, ...)>
   <!ATTLIST country    car_code ID #REQUIRED          memberships IDREFS #IMPLIED
                        capital IDREFS #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT city (name, population*)>      <!ATTLIST city country IDREF #REQUIRED >
<!ELEMENT population (#PCDATA)>       <!ATTLIST population year CDATA #IMPLIED>
<!ELEMENT organization (name, abbrev, members+)>
   <!ATTLIST organization    id ID #REQUIRED    seat IDREF #IMPLIED>
<!ELEMENT abbrev (#PCDATA)>
<!ELEMENT members EMPTY>
   <!ATTLIST members    type CDATA #REQUIRED    country IDREFS #REQUIRED>

<country car_code="B" capital="cty-Brussels" memberships="org-eu org-nato ...">
   <name>Belgium</name>
   <population>10170241</population>
   <city id="cty-Brussels" country="B">
      <name>Belgium</name>
      <population year="95">951580</population>
   </city>
   ⋮
</country>
<country car_code="D" capital="cty-Berlin" memberships="org-eu org-nato ...">
   ⋮
</country>
<organization id="org-eu" seat="cty-Brussels">
   <name>European Union</name>  <abbrev>EU</abbrev>
   <members type="member" country="GR F E A D I B L ..."/>
```

<members type="membership applicant" country="AL CZ ..."/>
</organization>

<organization id="org-nato" seat="cty-Brussels" ...>
⋮
</organization>

The XPath expression

//country[name]/city[population/text()>100000 and @zipcode]/name/text()

returns all names of cities such that the city belongs (i.e., is a subelement) to a country where a name subelement exists, the city's population is higher than 100000, and its zipcode is known.

XPath is only an addressing mechanism, not a full querying language like, e.g., the SQL querying construct. It provides the base for most XML querying languages, which extend it with their special constructs (e.g., functional style in XSLT, and SQL/OQL style (e.g., joins) in Quilt). In the case of XPathLog, the extension feature are Prolog/Datalog style variable bindings, joins, and rules. A first step towards variable bindings in XPath has been introduced in the 1998 XQL proposal [RLS98] (see Section 3.3) with *return operators*:

Example 5.2 (XQL Output Operators as Variables)
The following XPath expression with return operators returns pairs (x_1, x_2) *such that* x_1 *is the name of the country, and* x_2 *is the name of a city in this country:*

//country[name/text()?]//city[population/text() > 100000 and @zipcode]/name/text()?

Here, the following "variables" are used:

- *country[name/text()?] defines an unnamed return variable (position 1 of the return tuple),*
- *population/text() > 100000 defines an implicit local variable whose value is used only in the comparison,*
- *@zipcode defines a implicit* don't care variable *whose binding must be present but is never used,*
- */name/text()? defines another unnamed return variable (position 2 of the return tuple).*

In Logic Programming languages, instead of a result set, for every match, a *variable binding* of the free variables is returned which can be used in the rule head. We extend the XPath syntax with the Prolog-style variable concept (and with implicit dereferencing). First, a semi-formal intuitive definition, is given, extending XPath expressions. A formal definition based on the XPath grammar given in [XPa99, Chap. 2] will be presented in Definition 5.8.

Definition 5.1 (XPathLog: Syntax as derived from XPath)
An XPathLog *reference expression* is an XPath expression where

- every *location step* nodetest [*filter*]* may be replaced by
  - $variable_1$[*filter*]*,
  - nodetest→$variable_2$[*filter*]*,
  - nodetest[*filter*]*→$variable_2$ or even
  - $variable_1$[*filter*]*→$variable_2$ or
  - $variable_1$→$variable_2$[*filter*]*.

variable$_1$ is bound to the element name, extending the "*" navigation wildcard, and *variable*$_2$ is bound to the node.

- for reference attributes `@name`, additional navigation steps may follow (implicit dereferencing).
- the id function is not used (since XPathLog supports implicit dereferencing). □

The variables are bound to the names/nodes/literals (for i.e., `CDATA` or `NMTOKENS` attributes) which result from the respective match; the formal semantics based on that of XPath given in [Wad99b] (cf. Figure 3.1 in Section 3.2.1) is given in Definition 5.9. The following example illustrates the idea:

Example 5.3 (XPathLog: Introductory Queries)
The following examples are evaluated against the MONDIAL *database (an excerpt is given in Example 5.1).*

Pure XPath expressions: *pure XPath expressions (i.e., without variables) are interpreted as* existential queries *which return* true *if the result set is non-empty:*

```
?- //country[name/text() = "Belgium"]//city/name/text().
true
```

since the country element which has a name subelement with the text contents "Belgium" contains at least one city descendant with a name subelement with non-empty text contents.

Output Result Set: *The query "?- xpath→N" for any xpath binds N to all nodes belonging to the result set of* xpath*:*

```
?- //country[name/text() = "Belgium"]//city/name/text()→N.
N/"Brussels"
N/"Antwerp"
⋮
```

respectively, for a result set consisting of elements, logical ids are returned (here, using mnemonic ids instead of o_1, o_2 etc):

```
?- //country[name/text() = "Belgium"]//city→C.
C/brussels
C/antwerp
⋮
```

Additional Variables: *XPathLog allows to bind all nodes which are traversed by an expression (both by the access path to the result set, and in the filters):*
The following expression returns all tuples (N_1, C, N_2) such that the city with name N_2 belongs to the country with name N_1 and car code C:

```
?- //country[name/text()→N1 and @car_code→C]//city/name/text()→N2.
N2/"Brussels" C/"B" N1/"Belgium"
N2/"Antwerp" C/"B" N1/"Belgium"
    ⋮
N2/"Berlin" C/"D" N1/"Germany"
    ⋮
```

Local Variables: *The following XPath expression returns all names of cities such that the city belongs to a country whose name is known and its population is higher than 100000:*

```
?- //country[name/text()→N1]//city[population/text()→_P]/name/text()→N2,
     _P > 100000.
```

The semantics of this query is a set of variable bindings for N_1 and N_2.

Dereferencing: *For every organization, give the name of the seat city and all names and types of members:*

```
?- //organization[name/text()→N and abbrev/text()→A and @seat/name/text()→SN]
                       /members[@type→MT]/@country/name/text()→MN.
```

One element of the result set is e.g.,

N/"..." A/"EU" SN/"Brussels" MT/"member" MN/"Belgium"

Navigation Variables: *Are there elements which have a name subelement with the **PCDATA** contents "Monaco", and of which type are they?*

```
?- //Type→X[name/text()→"Monaco"].
     Type/country    X/country-monaco
     Type/city       X/city-monaco
```

Schema Querying: *The use of variables at name positions further allows for schema querying, e.g., to give all names of subelements of elements of type* city*:*

```
?- //city/SubElName.
SubElName/name
SubElName/population
⋮
```

The schema querying functionality can also be used for validation wrt. a DTD or a given XML Schema (which can be present as an XML tree in the same XPathLog database).

Recall that XPath does not support implicit dereferencing (cf. Section 3.2.2, Example 3.4), but uses the id(...) function which leads to confusing expressions if navigation is applied along several references.

Example 5.4 (Dereferencing)
The XPathLog expression

```
?- //organization[@seat = members/@country/@capital]/@seat/name/text()→N.
```

(all city names which are seats of an organization and the capital of one of its members) is equivalent to the XPath expression

```
id(//organization[id(./@seat) = id(id(./members/@country)/@capital)]/@seat)/name/text()
```

As described in Sections 3.10 and 3.11, an explicit dereferencing operator "→"-operator has been defined in Quilt/XQuery, which gives also the type of the referenced element [CRF00, XQu01]:

```
//organization[@seat→city = members/@country→country/@capital→city]
                /@seat→city/name/text() .
```

XPathLog additionally allows to add further variables to bind the names of the organization and of the country:

```
?- //organization[name/text()→ON and
                       @seat = members/@country[name/text()→CN]/@capital]
                     /@seat/name/text()→N.
```

Further examples can be found and executed with the LoPiX *system [LoP].*

5.2 Notation and Basic Notions

Logic-oriented frameworks are in general based on a semantical structure (from which a suitable Herbrand-style structure is derived) as a theoretical foundation. Since XPath-Logic is based on the notions of first-order logic, a short review of first-order logic is given, introducing the notation in this work. As a prerequisite, a syntax and semantics for handling mappings and especially lists are needed:

Notation 5.1 (Lists)
Throughout this work, the following notation is used:

- For two sets A and B, the set of mappings from A to B is denoted by B^A (cf. 2^A for the powerset of A, mapping each element of A to either 1 or 0 (contained or not)).
- A list over a domain D is a mapping from $\mathbb{N}$ to D. Thus, the set of lists over D is denoted by $D^{\mathbb{N}}$.
- the empty list is denoted by ε; a unary list containing only the element x is denoted by (x); list concatenation as an operator is denoted by $\circ$, e.g., $(c) \circ \ell$ where ℓ is a list.
- $\mathsf{append}(\ell, c)$ results in the list $\ell \circ (c)$.
- $\mathsf{set}(expr_1(x_1, \ldots, x_n) \mid expr_2(x_1, \ldots, x_n, i))$ stands for

  $$\{expr_1(x_1, \ldots, x_n) \mid expr_2(x_1, \ldots, x_n)\}$$

  In the following, sets are sometimes used as lists exploiting the fact that a set can be seen as a list by an arbitrary enumeration.
- In a similar way, a list can be constructed by enumerating its elements. For an enumerable domain I (set of indices; I can also be a list)

  $$\mathsf{list}_{i \in I}(expr_1(x_1, \ldots, x_n) \mid expr_2(x_1, \ldots, x_n, i))$$

  is the list $(expr_1(x_{1,1}, \ldots, x_{1,n}), expr_1(x_{2,1}, \ldots, x_{2,n}), \ldots,)$ such that the values $x_{i,1}, \ldots, x_{i,n}$ are defined by $expr_2(x_1, \ldots, x_n, i)$.
- Similar to list,

  $$\mathsf{concat}_{i \in I}(expr_1(x_1, \ldots, x_n) \mid expr_2(x_1, \ldots, x_n, i))$$

  does the same if $expr_1(x_1, \ldots, x_n)$ is already a list.
- For a finite list $\ell = (x_1, \ldots, x_n)$, $\mathsf{reverse}(\ell) = (x_n, \ldots, x_1)$.
- For a list ℓ, $\ell[i, j]$ denotes the sublist consisting of the ith to jth elements,
- For a list ℓ of pairs i.e., $\ell = ((x_1, y_1), (x_2, y_2), \ldots)$, $\ell\downarrow_1$ denotes the projection of the list to the first component of the list elements, i.e., $\ell\downarrow_1 := (x_1, x_2, \ldots)$. □

First-Order Logic. Each first-order language contains a set of distinguished symbols, consisting of parentheses "(" and ")", constants true, false representing the truth values, boolean connectives $\neg$, $\wedge$, $\vee$, quantifiers $\forall$, $\exists$, and an infinite set Var of variables x, y, $x_1, x_2, \ldots$.

An individual first-order language is then given by its *signature* Σ. Σ is partitioned into a functional part $\mathcal{F}$ of function symbols and a relational part $\mathcal{P}$ of predicate symbols, each of the symbols with a given arity which is denoted by $\mathsf{ord}(f)$ and $\mathsf{ord}(p)$, respectively. 0-ary functions are also called *constants*.

The set Term_Σ of *terms* over Σ is defined inductively as

- each variable is a term,
- for $f \in \mathcal{F}$, $\mathsf{ord}(f) = n$ and terms $t_1, \ldots, t_n$, also $f(t_1, \ldots, t_n)$ is a term.

The set of *atomic formulas* over Σ is given as

$$\mathsf{Atoms}_\Sigma := \{s = t \mid s, t \in \mathsf{Term}_\Sigma\} \cup \{p(t_1, \ldots, t_n) \mid p \in \mathcal{P}, \mathsf{ord}(p) = n, t_1, \ldots, t_n \in \mathsf{Term}_\Sigma\} .$$

The set of *first-order formulas* over Σ is defined as the least set with the following properties:

- all atomic formulas are formulas,
- true and false are formulas,
- for formulas A and B, a variable x, $\neg A$, $A \vee B$, $A \wedge B$, $\forall x : A$, and $\exists x : A$ are formulas.

The notions of bound and free variables are defined in the usual way, $\mathsf{free}(F)$ denoting the set of variables occurring free in a set F of formulas.
The semantics of first-order logic is given by *first-order structures* over a given signature:

Definition 5.2 (First-Order Structure)
A *first-order structure* $\mathcal{I} = (I, \mathcal{U})$ over a signature Σ consists of a nonempty set $\mathcal{U}$ (*universe*) and an *interpretation* I of the signature symbols over $\mathcal{U}$ which maps

- every constant c to an element $I(c) \in \mathcal{U}$,
- every n-ary function symbol f to an n-ary function $I(f) : \mathcal{U}^n \to \mathcal{U}$,
- every n-ary predicate symbol p to an n-ary relation function $I(p) : \mathcal{U}^n \to \{\mathsf{true}, \mathsf{false}\}$.

For short, I consists of two mappings $I_F : \mathcal{F} \to \bigcup_{k=0}^{\infty} \mathcal{U}^{(\mathcal{U}^k)}$ and $I_P : \mathcal{P} \to \bigcup_{k=1}^{\infty} 2^{(\mathcal{U}^k)}$ □

A *variable assignment* over a universe $\mathcal{U}$ is a mapping

$$\beta : \mathsf{Var} \to \mathcal{U} .$$

For a variable assignment β, a variable x, and $d \in \mathcal{U}$, the *modified* variable assignment β_x^d is identical with β except that it assigns d to the variable x:

$$\beta_x^d : \mathsf{Var} \to \mathcal{U} : \begin{cases} y \mapsto \beta(y) & \text{if } y \neq x \ , \\ x \mapsto d & \text{otherwise.} \end{cases}$$

Similar, for a variable assignment β and a variable x, $\beta \setminus \{x\}$ denotes β without the mapping for x:

$$\beta \setminus \{x\} : \mathsf{Var} \to \mathcal{U} : \begin{cases} y \mapsto \beta(y) & \text{if } y \neq x \ , \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Every structure $\mathcal{I}$ induces an evaluation $\mathcal{I}$ of terms

$$\mathcal{I} : \mathsf{Term}_\Sigma \times \text{Var\_Assignments} \to \mathcal{U}$$

and tuples of terms, $\mathcal{I} : \mathsf{Term}_\Sigma^n \times \text{Var\_Assignments} \to \mathcal{U}^n$, as follows:

$$\begin{array}{l}
\mathcal{I}(x, \beta) := \beta(x) \quad \text{for a variable } x \ , \\
\mathcal{I}((t_1, \ldots, t_n), \beta) := (\mathcal{I}(t_1, \beta), \ldots, \mathcal{I}(t_n, \beta)) \quad \text{for terms } t_1, \ldots, t_n \ , \\
\mathcal{I}(f(t_1, \ldots, t_n), \beta) := (I(f))(\mathcal{I}((t_1, \ldots, t_n), \beta)) = (I(f))(\mathcal{I}(t_1, \beta), \ldots, \mathcal{I}(t_n, \beta)) \\
\qquad \text{for a function symbol } f \in \Sigma, \mathsf{ord}(f) = n \text{ and terms } t_1, \ldots, t_n.
\end{array}$$

To indicate the truth of a formula F in a structure $\mathcal{I}$ under a variable assignment β, the standard notation $\models$ is used: Let s, t be terms, p a predicate symbol, $\mathsf{ord}(p) = n$, $t_1, \ldots, t_n$ terms, x a variable, A and B formulas. Then

$$\begin{array}{lll}
(\mathcal{I}, \beta) \models \mathsf{true}\ , & & \\
(\mathcal{I}, \beta) \models p(t_1, \ldots, t_n) & :\Leftrightarrow & (\mathcal{I}(t_1, \beta), \ldots, \mathcal{I}(t_n, \beta)) \in I(p)\ , \\
(\mathcal{I}, \beta) \models \neg A & :\Leftrightarrow & \text{not } (\mathcal{I}, \beta) \models A\ , \\
(\mathcal{I}, \beta) \models A \vee B & :\Leftrightarrow & (\mathcal{I}, \beta) \models A \text{ or } (\mathcal{I}, \beta) \models B\ , \\
(\mathcal{I}, \beta) \models \exists x : A & :\Leftrightarrow & \text{there is a } d \in \mathcal{U} \text{ with } (\mathcal{I}, \beta_x^d) \models A\ .
\end{array}$$

The symbols $A \wedge B := \neg(\neg A \vee \neg B)$, $A \to B := \neg A \vee B$ and $\forall x : F := \neg \exists x : \neg F$ are defined as usual.

We start with introducing X-structures as a semantical notion for XML instances, and then give the syntax of the particular logic.

5.3 XML Instances as Semantical Structures

When considering XML instances, we see an XML instance as an abstract structure over a signature $\Sigma = (\Sigma_N, \Sigma_F, \Sigma_C, \Sigma_P)$ which consists of

- Σ_N: element names and attribute names (termed `qnames` in XML),
- Σ_F: names of XML-built-in functions,
- Σ_C: constant symbols which denote elements in the XML instance (e.g., the constant germany may be interpreted as the element node addressed by /mondial/country[name="Germany"]).
- Σ_P: predicates (with arity).
- Additionally, a basic set of literals, i.e., strings and numbers is assumed.

In contrast to the DOM model [DOM98] (see Section 2.1.3) and the XML Query Data Model [XMQ01b] (see Section 3.9.1) which use a node-labeled tree (i.e., the element and attribute names are associated with the nodes), an *edge-labeled* model is used in the present approach. Using an edge-labeled model proves useful for data integration (see Section 11). The navigation graph induced by the data model is a variant of the semistructured data model defined in [Bun97]. Recall from Section 3.5 that XML-QL [DFF$^+$99b] also uses an *edge-labeled* graph which especially defines the same handling of text contents as ours. Their data model has been influenced by the experiences with the STRUDEL/STRUQL [FFK$^+$98] project for data integration.

The main features of the model are:

- the universe consists of the *element nodes* of the XML instance and *literals* used as attribute values and text contents;
- *element nodes* have properties, defined by (i) subelements (which are ordered) and (ii) attributes (which are unordered);
- multivalued attributes (`NMTOKENS` and `IDREFS`) are silently split;
- reference attributes are silently resolved.

This model supports a declarative handling of `IDREFS` attributes without the necessity of string operators.

Definition 5.3 (X-Structure)
An X-structure over a given signature Σ is a tuple

$$\mathcal{X} = (\mathcal{N}, \mathcal{V}, \mathcal{L}, \mathcal{I}, \mathcal{E}, \mathcal{A})$$

where

- $\mathcal{N}$ is the set of names which may be further distinguished as
  - $\mathcal{N}_{\mathcal{E}}$: element names (as e.g., occurring in node tests); $\mathcal{N}_{\mathcal{E}}$ contains a special element text() for handling text children.
  - $\mathcal{N}_{\mathcal{A}}$: attribute names,
  - later, also class names $\mathcal{N}_{\mathcal{C}}$ will be introduced which may further be distinguished as $\mathcal{N}_{\mathcal{CE}}$ and $\mathcal{N}_{\mathcal{CL}}$ for element and literal classes;
- $\mathcal{V}$ is a set of nodes (from the graph point of view, vertices), identified by internal names,
- $\mathcal{L}$ is a set of literals (integers, floats, strings),
- $\mathcal{I}$ is a (partial) mapping, which interprets the signature:

  $$\begin{array}{rcl} \mathcal{I}_{\mathcal{E}} & : & \Sigma_N \to \mathcal{N}_{\mathcal{E}} \\ \mathcal{I}_{\mathcal{A}} & : & \Sigma_N \to \mathcal{N}_{\mathcal{A}} \end{array}$$

  interpret the names Σ_N by element and attribute names (note that a name may simultaneously be an element name and an attribute name, e.g. <el n="foo"> <n> ...</n></el> is allowed);

  $$\mathcal{I}_C \ : \ \Sigma_C \to \mathcal{V}$$

  interprets the constant symbols in Σ_C by nodes in $\mathcal{V}$;

  $$\mathcal{I}_F \ : \ \mathcal{V} \times \Sigma_F \times (\mathcal{V} \cup \mathcal{L} \cup \mathcal{N})^* \to \mathcal{V} \cup \mathcal{L} \cup \mathcal{N}$$

  represents the interpretation of built-in functions, and

  $$\mathcal{I}_P \ : \ \Sigma_P \times (\mathcal{V} \cup \mathcal{L} \cup \mathcal{N})^* \to \{\mathit{true}, \mathit{false}\}$$

  represents the interpretation of predicates.
- $\mathcal{E}$ is a (partial) mapping

  $$\mathcal{E} \ : \ \mathcal{V} \times \mathcal{N}_{\mathcal{E}} \times \mathbb{N} \to \mathcal{V} \cup \mathcal{L}$$

  (subelement relationship; from the graph point of view, edges; ordered), and
- $\mathcal{A}$ is a (partial) mapping

  $$\mathcal{A} \ : \ \mathcal{V} \times \mathcal{N}_{\mathcal{A}} \to 2^{\mathcal{V}} \cup 2^{\mathcal{L}}$$

  (attribute values).

Note that $\mathcal{E}$ and $\mathcal{A}$ are not direct interpretations of Σ, but mappings that "interprete" $\mathcal{N}$. Σ is mapped to $\mathcal{N}$ before by $\mathcal{I}$, making attribute and element names full citizens of the language.
Furthermore, $\mathcal{E}$ is partitioned into the subelement part and the text contents part:

$$\mathcal{E}_{Sub}(v,n,i) := \begin{cases} \mathcal{E}(v,n,i) & \text{if } \mathcal{E}(v,n,i) \in \mathcal{L}\ , \\ \perp & \text{if } \mathcal{E}(v,n,i) \in \mathcal{V}\ , \end{cases}$$

$$\mathcal{E}_{text}(v,n,i) := \begin{cases} \mathcal{E}(v,n,i) & \text{if } \mathcal{E}(v,n,i) \in \mathcal{V}\ , \\ \perp & \text{if } \mathcal{E}(v,n,i) \in \mathcal{L}\ . \end{cases}$$

$\mathcal{A}$ is partitioned into the literal-valued part, and the reference part:

$$\mathcal{A}_{Lit}(v,n) := \begin{cases} \mathcal{A}(v,n) & \text{if } \mathcal{A}(v,n) \subseteq \mathcal{L} \ , \\ \perp & \text{if } \mathcal{A}(v,n) \subseteq \mathcal{V} \ , \end{cases}$$

$$\mathcal{A}_{Ref}(v,n) := \begin{cases} \mathcal{A}(v,n) & \text{if } \mathcal{A}(v,n) \subseteq \mathcal{V} \ , \\ \perp & \text{if } \mathcal{A}(v,n) \subseteq \mathcal{L} \ . \end{cases}$$

□

Note the following:

- *XML attribute nodes* do not belong to $\mathcal{V}$, but their literal values belong to $\mathcal{L}$.
- for reference attributes (`IDREF`), the "results" are not the `ID`-strings, but the target nodes themselves,
- `PCDATA` children are handled by the special name text(),

The elements of $\mathcal{V}$ (representing the element nodes) do not carry useful information in themselves, they are only of interest as anonymous entities (similar to object ids) which have certain properties that are given by $\mathcal{E}, \mathcal{A}$, and $\mathcal{I}$. In the following, mnemonic ids (e.g., *john*) or artificial ids (e.g., o_42) are used for elements of $\mathcal{V}$.

Definition 5.4 (Navigation Graph)
The *navigation graph* of an X-Structure is the directed graph $(\mathcal{V} \cup \mathcal{L}, \mathcal{E} \cup \mathcal{A})$ which is induced by all element nodes, the subelement relationship, and the attribute references. □

There is a canonical mapping from the set of XML instances to the set of X-structures (modulo equivalence):

Definition 5.5 (Canonical X-Structure)
Given an XML instance $\mathcal{D}$, the canonical X-structure $\mathcal{X}_\mathcal{D}$ is defined as follows:

- Starting with the root node, the document is traversed in document order, assigning an id with every element node (inducing a bijective mapping Φ from element nodes to ids). $\mathcal{V}$ is the set of used ids.
- the set Σ_N of names used in the XML instance (or in the DTD, if available) is mapped by the bijective mappings $\mathcal{I}_\mathcal{A}$ and $\mathcal{I}_\mathcal{E}$ to "internal names" in $\mathcal{N}_\mathcal{A}$ and $\mathcal{N}_\mathcal{E}$.
- The interpretations of names are defined as

$$\begin{array}{lcl} \mathcal{E}(v, name, i) & = & \Phi(node) \text{ where } node \text{ is the } i\text{th subelement of node } \Phi^{-1}(v) \text{ and} \\ & & \qquad \mathcal{I}_\mathcal{E}(\mathsf{name}(node)) = name \text{ (thus, covering only non-text elements).} \\ \mathcal{E}(v, \mathsf{text}(), i) & = & \Phi(node) \text{ where } node \text{ is the } i\text{th subelement of node } \Phi^{-1}(v) \\ & & \qquad \text{and } node \text{ is a text element.} \\ \mathcal{A}(v, name) & = & \begin{cases} \{\Phi(node) \mid node \text{ is a target node of the } \mathcal{I}_\mathcal{A}^{-1}(name) \text{ attribute of } \Phi^{-1}(v)\} \\ \qquad \text{if } \mathcal{I}_\mathcal{A}^{-1}(name) \text{ is a reference attribute of } \Phi^{-1}(v) \ , \\ \{\ell \mid \ell \text{ is a value of the } \mathcal{I}_\mathcal{A}^{-1}(name) \text{ attribute of } \Phi^{-1}(v)\} \\ \qquad \text{if } \mathcal{I}_\mathcal{A}^{-1}(name) \text{ is a non-reference attribute of } \Phi^{-1}(v) \ . \end{cases} \end{array}$$

  Interpretation of the name function:

$$\mathcal{I}(v, \mathsf{name}, \varepsilon) = \begin{cases} \mathcal{I}_\mathcal{E}(\mathsf{name}(\Phi^{-1}(v))) & \text{if } \Phi^{-1}(v) \text{ is an element node} \\ \mathsf{text}() & \text{if } \Phi^{-1}(v) \text{ is a text node.} \end{cases}$$

  The definitions of built-in functions in XPath contribute to $\mathcal{I}$. □

In the following, Σ_N is identified with $\mathcal{N}_\mathcal{E}$ and $\mathcal{N}_\mathcal{A}$, omitting $\mathcal{I}_\mathcal{E}$ and $\mathcal{I}_\mathcal{A}$.

Literals: Numbers and Strings. The basic XML data model does not distinguish between literal datatypes such as numbers and strings: in the DTD, both are text, as `PCDATA` (children) or as `CDATA` or `NMTOKEN` (attributes); also the ASCII representation does not distinguish them. XPath casts text data into numbers when required by the queries (e.g., for comparisons or arithmetics). In contrast XML Schema and the XML Query Data Model support distinguished literal datatypes. In this work, we follow the basic approach:

- if no datatype information is available, "numeric" data is treated as numbers (casted into strings if needed for string operations), and non-numeric data is treated as strings (here, the only problems are e.g., phone numbers which are "numeric" but may start with a leading "0").
- if datatype information is available, it is used.

Example 5.5 (X-Structure)
The XML instance given in Example 5.1 is represented by the following X-structure (using mnemonic ids); its graphical representation is given in Figure 5.1:

$\mathcal{N}_\mathcal{E}$ = {mondial, country, name, city, population, abbrev, members} ,
$\mathcal{N}_\mathcal{A}$ = {car_code, capital, memberships, id, seat, type, country} ,

$\mathcal{E}$(*mondial*,country,42) = *belgium*,
$\mathcal{A}$(*belgium*,car_code) = {"B"}, $\mathcal{A}$(*belgium*,capital) = {*brussels*},
$\mathcal{A}$(*belgium*,memberships) = {*eu*,*nato*,...}
$\mathcal{E}$(*belgium*,name,1) = *belgium-name*, $\mathcal{E}$(*belgium*,name,2) = *belgium-pop*,
$\mathcal{E}$(*belgium*,city,3) = *brussels*, $\mathcal{E}$(*belgium*,city,4) = ...,
$\mathcal{E}$(*belgium-name*,text(),1) = "Belgium", $\mathcal{E}$(*belgium-pop*,text(),1) = 10170241,
$\mathcal{A}$(*brussels*,country) = {*belgium*}, $\mathcal{A}$(*brussels*,id) = {"city-brussels"},
$\mathcal{E}$(*brussels*,name,1) = *brussels-name*, $\mathcal{E}$(*brussels*,population,2) = *brussels-pop*,
$\mathcal{E}$(*brussels-name*,text(),1) = "Brussels", $\mathcal{E}$(*brussels-pop*,text(),1) = 951580,

$\mathcal{E}$(*mondial*,country,45) = *germany*,
$\mathcal{A}$(*germany*,car_code) = {'D"}, $\mathcal{A}$(*germany*,capital) = {*berlin*},
$\mathcal{A}$(*germany*,memberships) = {*eu*,*nato*,...}

$\mathcal{E}$(*mondial*,organization,179) = *eu*,
$\mathcal{A}$(*eu*,id) = {"org-eu"}, $\mathcal{A}$(*eu*,seat) = {*brussels*},
$\mathcal{E}$(*eu*,name,1) = *eu-name*, $\mathcal{E}$(*eu-name*,text(),1) = "European Union",
$\mathcal{E}$(*eu*,abbrev,2) = *eu-abbrev*, $\mathcal{E}$(*eu-abbrev*,text(),1) = "EU",
$\mathcal{E}$(*eu*,members,3) = *eu-mem-mem*,
$\mathcal{A}$(*eu-mem-mem*,type) = {"member"}, $\mathcal{A}$(*eu-mem-mem*,country) = {*belgium*, *germany*, ...},
$\mathcal{E}$(*eu*,members,4) = *eu-mem-appl*,
$\mathcal{A}$(*eu-mem-appl*,type) = {"mem. appl."}, $\mathcal{A}$(*eu-mem-appl*,country) = {*albania*,...}.

$\mathcal{E}$(*mondial*,organization,180) = *nato*,
$\mathcal{A}$(*nato*,id) = {"org-nato"}, $\mathcal{A}$(*nato*,seat) = {*brussels*},

Note that the X-Structure contains references instead of id strings for `IDREF` attributes.

Remark 5.1
Note that the above definition depends on the knowledge about attribute declarations as `NMTOKENS`, `IDREF`, and `IDREFS` for splitting and resolving. Concerning reference attributes, the problem also occurs for all other XML querying languages – for resolving references via the id(.) function or XQuery's "→" operator, is must be known which attributes are declared as `ID`. Wrt. splitting `NMTOKENS`, the problem does not occur for other approaches, since they do not split them, but require explicit string operations for accessing individual values (cf. Section 3.2.2). □

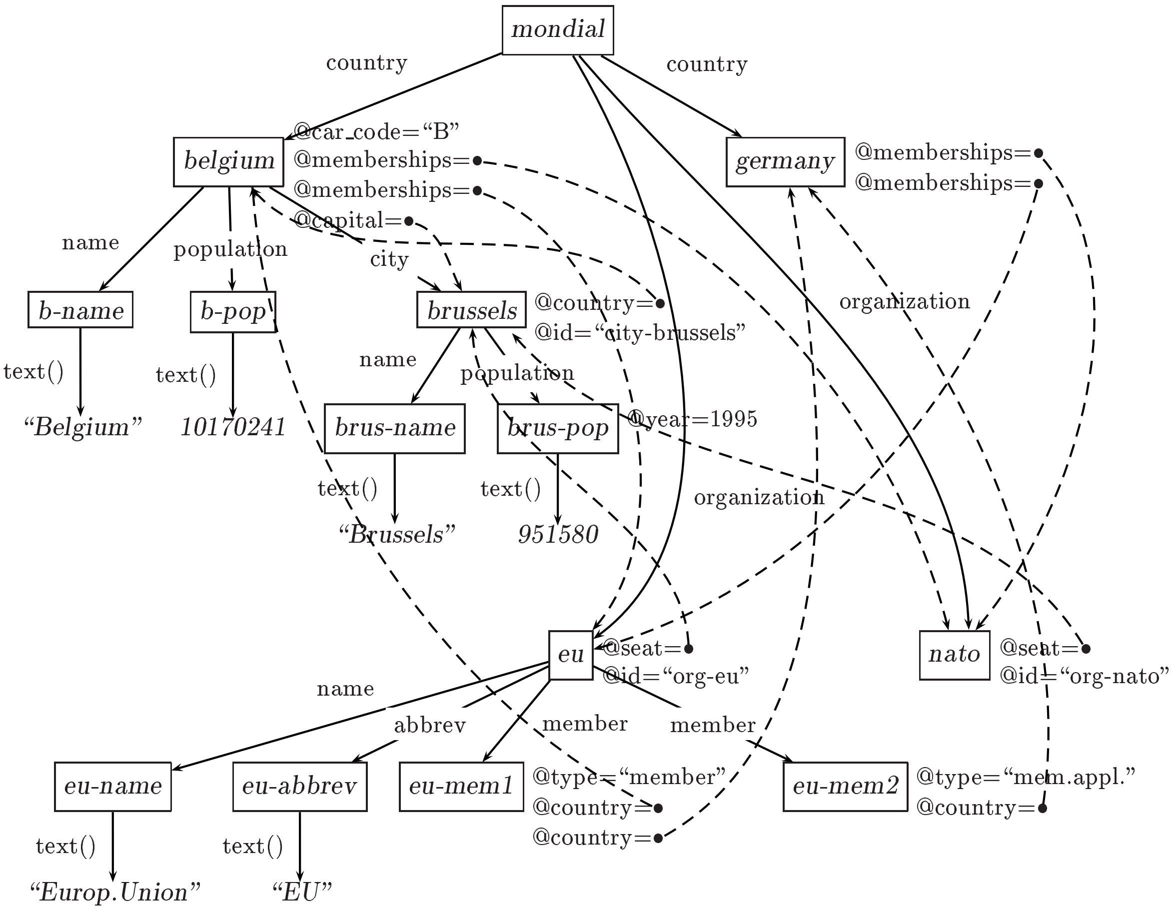

Figure 5.1: Example X-Structure

The canonical X-structure to an XML instance represents a single XML tree. Similar, several XML instances can be represented in an X-structure by *separate* trees, covering the DOM model.

In full generality, an X-structure can also contain subelement edges and reference edges which are not conforming with the XML tree model, but which are crucial for data integration (see Section 11):

- an element may be a subelement of several other elements, even with different names of the subelement relationship ("overlapping trees", cf. Figure 10.1),
- the subelement relationship may be cyclic (needed for restructuring),
- reference attributes may (temporarily) connect different trees.

Thus, an X-structure is not one or more "XML trees", but an *XML database* which contains simultaneously many views (each of which is an XML tree). When such a view should be regarded as a result tree, there are some requirements. Each node v of an X-structure is a potential root element for a result tree view which recursively consists of v's subelements and attributes. The following definition characterizes when the subtree rooted in v is a "useful" XML instance:

Definition 5.6 (Descendants and roots in an X-structure)
For an $\mathcal{X}$-structure, let

$$\begin{array}{rl} \mathsf{desc}(r,v) :\Leftrightarrow & \mathcal{E}(r,n,i) = v \text{ for some } n \text{ and some } i, \text{ or} \\ & \mathcal{E}(r,n,i) = x \text{ for some } n \text{ and some } i, \text{ and } \mathsf{desc}(x,v) \text{ holds} \end{array} .$$

An element r is a *root* in $\mathcal{X}$ if

- if $\mathsf{desc}(r,v)$ holds for some $v \in \mathcal{V}$ then, $\mathsf{desc}(v,v)$ does not hold, i.e., the descendant relation in the subtree rooted in r is acyclic, and
- if $\mathsf{desc}(r,v)$ and $v' \in \mathcal{A}_{Ref}(v,n)$ hold for some $n \in \mathcal{N}_{\mathcal{A}}$ and some $v,v' \in \mathcal{V}$, also $\mathsf{desc}(r,v')$ holds, i.e., the target of the reference also belongs to the subtree rooted in r. □

In the above definition, the view trees are only defined by the subelement relationship. A refined definition of *view trees* which also constrains the types of subelements and attributes that belong to the view will be given in Section 11.1 for practical use in data integration.

In the following, X-structures serve for defining a semantics for XPath-Logic, using XPath-like terms. Thus, the notion of *axes* as defined in the XML data model has to be mapped to X-structures:

Definition 5.7 (Basic Result Sets: Axes)
For every node x in an X-structure $\mathcal{X}$ and every *axis* a of the XML data model,

$$\mathcal{A}_{\mathcal{X}}(a,x) \in ((\mathcal{V} \cup \mathcal{L}) \times \mathcal{N})^{\mathbb{N}}$$

is the list of pairs $(value, name)$ generated by axis a with x as context node (do not confuse $\mathcal{A}_{\mathcal{X}}$ with $\mathcal{A}$ which denotes the interpretation of attributes in $\mathcal{X}$).

$$\begin{array}{lcl} \mathcal{A}_{\mathcal{X}}(\mathsf{child},x) & := & \mathsf{list}_{i\in\mathbb{N}}((y,name) \mid \mathcal{E}(x,name,i) = y) \\ \mathcal{A}_{\mathcal{X}}(\mathsf{attribute},x) & := & \mathsf{list}((y,name) \mid y \in \mathcal{A}(x,name)) \quad \text{by some enumeration.} \end{array}$$

For the other axes, $\mathcal{A}_{\mathcal{X}}(a,x)$ is derived from $\mathcal{A}_{\mathcal{X}}(\mathsf{child},x)$ according to the XML specification:

$$\begin{array}{lcl} \mathcal{A}_{\mathcal{X}}(\mathsf{parent},x) & := & \mathsf{set}((p,\mathcal{I}(p,\mathsf{name},())) \mid \ x \in \mathcal{A}_{\mathcal{X}}(\mathsf{child},p)\downarrow_1) \\ \mathcal{A}_{\mathcal{X}}(\mathsf{preceding\text{-}sibling},x) & := & \\ & & \mathsf{concat}_{p\in\mathcal{A}_{\mathcal{X}}(\mathsf{parent},x)\downarrow_1}(\mathsf{reverse}(\mathcal{A}_{\mathcal{X}}(\mathsf{child},p)[1,i{-}1]) \mid x = \mathcal{A}_{\mathcal{X}}(\mathsf{child},p)[i]) \\ \mathcal{A}_{\mathcal{X}}(\mathsf{following\text{-}sibling},x) & := & \\ & & \mathsf{concat}_{p\in\mathcal{A}_{\mathcal{X}}(\mathsf{parent},x)\downarrow_1}(\mathcal{A}_{\mathcal{X}}(\mathsf{child},p)[i{+}1,\mathsf{last}()] \mid x = \mathcal{A}_{\mathcal{X}}(\mathsf{child},p)[i]) \\ \mathcal{A}_{\mathcal{X}}(\mathsf{ancestor},x) & := & \mathsf{concat}_{(p,n)\in\mathcal{A}_{\mathcal{X}}(\mathsf{parent},x)}(((p,n)) \ \circ \ \mathcal{A}_{\mathcal{X}}(\mathsf{ancestor},p)) \\ \mathcal{A}_{\mathcal{X}}(\mathsf{descendant},x) & := & \mathsf{concat}_{(c,n)\in\mathcal{A}_{\mathcal{X}}(\mathsf{child},x)}((c,n) \ \circ \ \mathcal{A}_{\mathcal{X}}(\mathsf{descendant},c)) \end{array}$$

□

Remark 5.2
Note that in contrast to the XML/XPath data model and the semantics given in [Wad99b],

- $\mathcal{A}_{\mathcal{X}}$ does not return a list of nodes/literals, but a list of pairs $(node/literal, name)$. This allows for overlapping trees (cf. Section 11) where a node is a child of several parents, possibly by different labels.
- As a consequence, the $\mathsf{name}()$ function is not necessarily needed for node tests. □

In case that there are no overlapping trees, $\mathcal{A}_{\mathcal{X}}(axis, v)$ enumerates the nodes on axis *axis* wrt. the node v:

Proposition 5.1 (Axes in X-structures)
Let $\mathcal{X}_\mathcal{D}$ be the canonical X-structure to an XML instance $\mathcal{D}$ (then, there are no overlapping trees, i.e., there is a unique parent for each element). Let Φ as defined in Definition 5.5.

Then, for each $v \in \mathcal{V}$ and every axis $axis$ except the attribute axis, $\mathcal{A}_{\mathcal{X}_\mathcal{D}}(axis, x) \downarrow_1$ enumerates all nodes on the axis $axis$ starting from $\Phi^{-1}(x)$ in $\mathcal{D}$.

For the attribute axis, $\mathcal{A}_{\mathcal{X}_\mathcal{D}}(\mathsf{attribute}, x)$ does not enumerate the attribute *nodes* in the DOM tree, but (i) silently splits `NMTOKENS` attributes, and (ii) resolves reference attributes. □

In case of overlapping trees, there may several parents, and thus, also the sibling and ancestor axes get a different semantics, enumerating all siblings wrt. each parent. If namespaces are used for distinguishing these trees (cf. Section 11.4), there is in practice at most one parent for every namespace. In the meantime, assume that there is a unique parent for simplicity.

5.4 Syntax of XPath-Logic

Inspired by first-order logic as a logic for dealing with first-order structures, XPath-Logic is defined for expressing properties of XML-Structures. The main difference between XPath-Logic and first-order logic is that XPath-Logic has an additional type of atomic formulas: *reference expressions* which turn out to be similar to predicates. XPathLog is then the Horn fragment of XPath-Logic. The "basic" components of the language are XPath-Logic *LocationPaths* which are syntactically derived from XPath's *LocationPaths*. In the following, a formal definition is given based on the W3C XPath [XPa99] specification which complements the informal motivation given in Section 5.1.

Definition 5.8 (XPath-Logic: Syntax)
The set of basic formulas of an XPath-Logic language is defined over a signature Σ as above, consisting of names (element names, attribute names, function names, constant symbols, and predicate names (playing only a minor role for integrating first-order logic)):

- every language contains an infinite set Var of variables.
- *XPath-Logic reference expressions* over the above names extend the XPath syntax as follows (referring to the definition numbering in [XPa99]):
  - XPath-Logic *reference expressions* are either XPath *AbsoluteLocationPaths* [XPa99, Chap. 2], or they are location paths which start at nodes of the universe that are given as constants or bound to variables (note that wrt. X-structures, an *AbsoluteLocationPath* is also simply a path which starts with the distinguished root element):

```
[0] ReferenceExpression ::= AbsoluteLocationPath
                          | ConstantLocationPath
[2b] ConstantLocationPath ::= constant "/" RelativeLocationPath
                            | variable "/" RelativeLocationPath
```

  - In XPath-Logic *LocationSteps*, *axis*::*name*[*filter*]* may be extended to bind the selected nodes to variables:

```
[4] Step ::= AxisSpecifier "::" NodeTest Predicate*
           | AxisSpecifier "::" NodeTest Predicate* "->" Var Predicate*
           | AxisSpecifier "::" Var Predicate*
           | AxisSpecifier "::" Var Predicate* "->" Var Predicate*
```

  where `NodeTest` is either a name in Σ_N, or one of the type tests text() or node() from XPath.

  – If *predExpr* is an XPath-Logic predicate expression, then also $\exists x : predExpr$ and $\forall x : predExpr$ are XPath-Logic predicate expressions:

```
[9] PredicateExpr ::= Expr
                    | exists Var ":" "(" Expr ")"
                    | forall Var ":" "(" Expr ")"
```

  – Note that by rules [4], [5], and [6] of [XPa99], expressions of the form `a/@b/c` are allowed. The XPath semantics given in [Wad99b] (see also Section 3.2.1 and Figure 3.1) evaluates any navigation starting from attribute values to false. In our semantics, this syntax denotes navigation by dereferencing `IDREF` attributes.

- An *XPath-Logic predicate* is
  – an XPath-Logic reference expression (which evaluates to true if the result set of the reference expression is non-empty), or
  – a predicate over XPath-Logic reference expressions.
- XPath-Logic Formulas are built over predicates and reference expressions, using $\wedge$, $\vee$, $\neg$, $\exists$, and $\forall$. □

5.5 Semantics

The semantics of XPathLogic is defined similar to first-order logic by structural induction. The main task here is to define the semantics of reference expressions, handling navigation, order, and filtering. A reference expression simultaneous acts as a term (it has a result (set) and can be compared to terms) and as a predicate (when used in a filter). Especially, reference expressions are allowed to contain formulas in the filters.

Remark 5.3 (Order)
Recall that the result sets of XPath expressions are *unordered* [XPa99]:

```
The primary syntactic construct in XPath is the expression.
An expression matches the production Expr. An expression is
evaluated to yield an object, which has one of the following
four basic types:

- node-set (an unordered collection of nodes without duplicates)
- boolean (true or false)
- number (a floating-point number)
- string (a sequence of UCS characters)
```

Only internally to the computation, when single steps are considered, there is a *current node list* which is used for applying the proximity position predicates (i.e., filters of the form [position()=i]; abbreviated by [i], or [last()]).

Although, most implementations return the elements of the result set in document order. Similar to the implementations, our semantics is *list-based* – using the fact that *lists* are a specialization of *(multi)sets*. □

The basic result lists are provided by $\mathcal{A}_{\mathcal{X}}(axis, v)$ for every node v of $\mathcal{X}$ (cf. Definition 5.7). Recall that for $\mathcal{A}_{\mathcal{X}}(\mathsf{attribute}, x)$, the result set contains literals in case of non-reference attributes, and element nodes in case of reference attributes. $\mathcal{A}_{\mathcal{X}}(\mathsf{attribute}, x)$ has been defined as a list using an arbitrary (but fixed) enumeration.

5.5.1 Semantics of Expressions

Expressions are decomposed into their *location steps.* Every location step consists of choosing an axis, preselecting nodes by a *node test*, and filtering the result by (i) "normal" predicates and (ii) *proximity position predicates* which use the order of the intermediate result set for selecting a certain element by its index.

Definition 5.9 (Semantics of XPath-Logic expressions)
The semantics is defined by operators $\mathcal{S}$ and $\mathcal{Q}$ which are derived from the formal semantics given in [Wad99b] (cf. Figure 3.1 in Section 3.2.1).

- $\mathcal{S}_\mathcal{X}$: Reference_Expressions $\rightarrow (\mathcal{V} \cup \mathcal{L} \cup \mathcal{N})^{\mathbb{N}}$,
  (Reference_Expressions $\times$ Var_Assignments) $\rightarrow (\mathcal{V} \cup \mathcal{L} \cup \mathcal{N})^{\mathbb{N}}$,
  (Axes $\times$ Reference_Expressions $\times \mathcal{V} \times$ Var_Assignments) $\rightarrow (\mathcal{V} \cup \mathcal{L} \cup \mathcal{N})^{\mathbb{N}}$

  evaluates reference expressions wrt. an axis, a context node, and a variable assignment and returns a result list. In the third case, we use $\mathcal{S}^{any}$ to denote that the actual value of $axis$ does not matter. Arithmetic expressions are also reference expressions.

- $\mathcal{Q}_\mathcal{X}$: (Predicate_Expressions $\times \mathcal{V} \times$ Var_Assignments) $\rightarrow$ Boolean

  evaluates filter expressions wrt. a context node and a variable assignment.

- Recall that $\mathcal{I}_F$ and $\mathcal{I}_P$ provide an interpretation of XPath built-in functions and predicates according the the specification [XPa99].

1. for closed expressions (i.e., without free variables),

$$\mathcal{S}_\mathcal{X}(\mathit{refExpr}) = \mathcal{S}_\mathcal{X}(\mathit{refExpr}, \emptyset)$$

2. reference expressions are translated into location paths wrt. a start node:

   - entry points: rooted path:

$$\mathcal{S}_\mathcal{X}(/p, \beta) = \mathcal{S}^{any}_\mathcal{X}(p, root, \beta)$$

   - entry points: constants c:

$$\mathcal{S}_\mathcal{X}(c/p, \beta) = \mathcal{S}^{any}_\mathcal{X}(p, \mathcal{I}_C(c), \beta)$$

   (this is mainly of interest in Section 11, when multiple documents are used; but also in the core XPathLog language, names/identifiers may be associated to nodes, see Section 7.2)

   - entry points: variables $v \in \mathsf{Var}$:

$$\mathcal{S}_\mathcal{X}(v/p, \beta) = \mathcal{S}^{any}_\mathcal{X}(p, \beta(v), \beta)$$

3. location step:

$$\mathcal{S}^{any}_\mathcal{X}(axis :: pattern, x, \beta) = \mathcal{S}^{axis}_\mathcal{X}(pattern, x, \beta)$$

   where *pattern* is of the form *nodetestremainder* where *remainder* is a sequence of filters and variable bindings. These are evaluated left to right, always applying the rightmost "operation" (filter or variable) to the result of the left part:

4. node test:

$$\begin{array}{lcl}
\mathcal{S}^a_\mathcal{X}(name, x, \beta) & = & \mathsf{list}_{(v,n)\in\mathcal{A}_\mathcal{X}(a,x)}(v \mid n = name) \\
\mathcal{S}^a_\mathcal{X}(\mathsf{node}(), x, \beta) & = & \mathsf{list}_{(v,n)\in\mathcal{A}_\mathcal{X}(a,x)}(v \mid v \in \mathcal{V}) \\
\mathcal{S}^a_\mathcal{X}(\mathsf{text}(), x, \beta) & = & \mathsf{list}_{(v,n)\in\mathcal{A}_\mathcal{X}(a,x)}(v \mid v \in \mathcal{L}) \\
\mathcal{S}^a_\mathcal{X}(N, x, \beta) & = & \mathsf{list}_{(v,n)\in\mathcal{A}_\mathcal{X}(a,x)}(v \mid n = \beta(N))
\end{array}$$

5. step with variable:

$$\mathcal{S}^a_{\mathcal{X}}(pattern \to V, x, \beta) \quad = \quad \begin{cases} (\beta(V)) & \text{if } (\beta(V)) \in \mathcal{S}^a_{\mathcal{X}}(pattern, x, \beta) \\ \varepsilon & \text{otherwise.} \end{cases}$$

6. filter(s) (note that this can be either in the form *axis*::*nodetest*[*filter*]* in a location step, or (*refExpr*)[*filter*]*):

$$\mathcal{S}^a_{\mathcal{X}}(pattern[\mathit{filter}], x, \beta) = \mathsf{list}_{y \in \mathcal{S}^a_{\mathcal{X}}(pattern,x,\beta)}(y \mid \mathcal{Q}_{\mathcal{X}}(\mathit{filter}, y, \beta^{k,n}_{\mathit{Pos,Size}}))$$

where $L_1 := \mathcal{S}^a_{\mathcal{X}}(pattern, x, \beta)$ and $n := size(L_1)$, and for every y, let j the index of y in L_1, $k := j$ if a is a forward axis, and $k := n{+}1{-}j$ if a is a backward axis (cf. [Wad99b]). Pos and $Size$ are only used if the filter contains a proximity position predicate.

7. Path:

$$\mathcal{S}^a_{\mathcal{X}}(p_1/p_2, x, \beta) = \mathsf{concat}_{y \in \mathcal{S}^a_{\mathcal{X}}(p_1,x,\beta)}(\mathcal{S}^{any}_{\mathcal{X}}(p_2, y, \beta))$$

Filters

8. Reference expressions (existential semantics) in filters:

$$\mathcal{Q}_{\mathcal{X}}(\mathit{refExpr}, y, \beta) \;:\Leftrightarrow\; \mathcal{S}^{any}_{\mathcal{X}}(\mathit{refExpr}, y, \beta) \neq \emptyset$$

9. Predicate expressions:

$$\begin{aligned} &\mathcal{Q}_{\mathcal{X}}(pred(expr_1, \ldots, expr_n), y, \beta) \;:\Leftrightarrow \\ &\qquad \text{there are } x_1 \in \mathcal{S}^{any}_{\mathcal{X}}(expr_1, y, \beta), \ldots, x_n \in \mathcal{S}^{any}_{\mathcal{X}}(expr_n, y, \beta) \\ &\qquad \text{such that } (x_1, \ldots, x_n) \in \mathcal{I}_P(pred) \end{aligned}$$

Recall that $\mathcal{I}_P$ is an interpretation of predicates, including built-in predicates of XPath.

10. Boolean Connectives:

$$\begin{aligned} \mathcal{Q}_{\mathcal{X}}(A \text{ and } B, y, \beta) &\;:\Leftrightarrow\; \mathcal{Q}_{\mathcal{X}}(A, y, \beta) \text{ and } \mathcal{Q}_{\mathcal{X}}(B, y, \beta) \\ \mathcal{Q}_{\mathcal{X}}(A \text{ or } B, y, \beta) &\;:\Leftrightarrow\; \mathcal{Q}_{\mathcal{X}}(A, y, \beta) \text{ or } \mathcal{Q}_{\mathcal{X}}(B, y, \beta) \\ \mathcal{Q}_{\mathcal{X}}(\text{not } A, y, \beta) &\;:\Leftrightarrow\; \text{not } \mathcal{Q}_{\mathcal{X}}(A, y, \beta) \end{aligned}$$

11. Quantification

$$\begin{aligned} \mathcal{Q}_{\mathcal{X}}(\exists X : A, y, \beta) &\;:\Leftrightarrow\; \text{there is an } x \in \mathcal{V} \cup \mathcal{L} \cup \mathcal{N} \text{ s.t. } \mathcal{Q}_{\mathcal{X}}(A, y, \beta^x_X) \\ \mathcal{Q}_{\mathcal{X}}(\forall X : A, y, \beta) &\;:\Leftrightarrow\; \text{for all } x \in \mathcal{V} \cup \mathcal{L} \cup \mathcal{N},\ \mathcal{Q}_{\mathcal{X}}(A, y, \beta^x_X) \end{aligned}$$

Comparisons.

12. The semantics of comparisons in XPath is *element-oriented*: E.g., *refExpr* = *term* evaluates to *true* if the result set of *refExpr* contains at least one element which equals *term*.

$$\begin{aligned} &\mathcal{Q}_{\mathcal{X}}(expr_1 \;\mathsf{op}\; expr_2, y, \beta) \;:\Leftrightarrow \\ &\qquad \text{there are } x_1 \in \mathcal{S}^{any}_{\mathcal{X}}(expr_1, y, \beta) \text{ and } x_2 \in \mathcal{S}^{any}_{\mathcal{X}}(expr_2, y, \beta) \text{ such that } x_1 \;\mathsf{op}\; x_2 \end{aligned}$$

where $\mathsf{op} \in \{<, \leq, >, \geq, =, \neq\}$.

Evaluation of Terms

13. constants (in free predicates or in comparisons in filters); let *value* be a string or a number:

$$\mathcal{S}^a_{\mathcal{X}}(value, x, \beta) = value$$

For constants $c \in \Sigma_C$:

$$\mathcal{S}^a_{\mathcal{X}}(c, x, \beta) = \mathcal{I}_C(c)$$

14. variables (in free predicates or in comparisons in filters);

$$\mathcal{S}^a_{\mathcal{X}}(var, x, \beta) = \beta(var)$$

15. functions:

$$\mathcal{S}^a_{\mathcal{X}}(f(expr_1, \ldots, expr_n), x, \beta) = \mathcal{I}_F(\beta(x), f, (\mathcal{S}^{any}(expr_1, x, \beta), \ldots, \mathcal{S}^{any}(expr_n, x, \beta)))$$

Recall that $\mathcal{I}_F$ is an interpretation of functions, including built-in functions of XPath. Note also that built-in functions (e.g., $\mathsf{count}(expr)$) regard $\mathcal{S}^{any}(expr, x, \beta)$ as a set.

16. context-related functions use the extension of variable bindings by pseudo-variables $Size$ and Pos in rule (6):

$$\mathcal{S}^{any}_{\mathcal{X}}(\mathsf{position}(), x, \beta) = \beta(Pos)$$
$$\mathcal{S}^{any}_{\mathcal{X}}(\mathsf{last}(), x, \beta) = \beta(Size)$$

Recall that the filter *pattern*[i] where $i \in \mathbb{N}$ stands for *pattern*[position()=i], thus, also *pattern*[last()] stands for *pattern*[position()=last()].

17. arithmetics:

$$\mathcal{S}^{any}_{\mathcal{X}}(expr_1 \;\mathsf{op}\; expr_2, x, \beta) = \mathsf{set}_{x_1 \in \mathcal{S}^{any}_{\mathcal{X}}(expr_1, x, \beta), x_2 \in \mathcal{S}^{any}_{\mathcal{X}}(expr_2, x, \beta)}(x_1 \;\mathsf{op}\; x_2)$$

The result is a set of literals which are not ordered anyhow (even not by the document order since the literals do not occur in the document). An arbitrary enumeration of the set is returned as result list.

Handling Result Sets.

18. Alternative/Union

$$\mathcal{S}^a_{\mathcal{X}}(p_1 | p_2, x, \beta) = \mathcal{S}^a_{\mathcal{X}}(p_1 \;\mathsf{union}\; p_2, x, \beta) = \mathcal{S}^a_{\mathcal{X}}(p_1, x, \beta) \circ \mathcal{S}^a_{\mathcal{X}}(p_2, x, \beta)$$ □

The semantics is comparable to XPath only for variable-free expressions. In a closed XPath-Logic reference expression, variables occur in the following situations:

- purely existential, i.e., the variable occurs once. If this occurrence is of the form $\exists var : path \to var$, then it can be equivalently replaced by $path$ in XPath. If the occurrence is in a predicate, e.g., $\exists var : p(\ldots, var, \ldots)$, then there is no equivalent variable-free XPath expression.
- as a join variable. Then, there is *sometimes* an equivalent XPath filter which can be very complex, especially if there are dependent variables. Thus, the existence of variables adds expressive power already for queries.

The following theorem states the equivalence of our semantics with the "official one" given in [Wad99b]:

Theorem 5.2 (Correctness of $\mathcal{S}$ and $\mathcal{Q}$ wrt. XPath)
For variable-free absolute reference expressions (i.e., pure XPath expressions) without

- navigating along reference attributes,
- splitting `NMTOKENS` attributes, and
- using the alternative construct "|" (which requires to merge the result lists using a global order),

the semantics coincides with the one given in [Wad99b]: For every such XPath expression *expr*,

$$\mathcal{S}_{\mathcal{X}}(expr) = \mathcal{S}[[expr]](x)$$

(for arbitrary x) where $\mathcal{S}[[expr]]$ is as defined in [Wad99b] and enumerated wrt. document order. □

The proof uses the following Lemma which contains the structural induction.

Lemma 5.3 (Correctness of $\mathcal{S}$ and $\mathcal{Q}$ wrt. XPath: Structural Induction)
The variable-free fragment of XPathLog without navigating along reference attributes corresponds to XPath as follows:

1. For absolute expressions (i.e., $expr = /expr'$), for all β:

   $$\mathcal{S}_{\mathcal{X}}(expr, \beta) = \mathcal{S}[[expr]](x)$$

   for arbitrary x.
2. For expressions, for all β:

   $$\mathcal{S}_{\mathcal{X}}(expr, v, \beta) = \mathcal{S}[[expr]](v) \ .$$
3. For filters, for all β:

   $$\mathcal{Q}_{\mathcal{X}}(\mathit{filter}, v, \beta^{k,n}_{Pos,Size}) \ \Leftrightarrow\ \mathcal{Q}[[\mathit{filter}]](v, k, n) \ .$$
4. For arithmetic expressions and built-in functions, for all β:

   $$\mathcal{S}_{\mathcal{X}}(expr, v, \beta^{k,n}_{Pos,Size}) = \mathcal{E}[[expr]](v, k, n) \ .$$

where $\mathcal{Q}[[expr]]$, $\mathcal{S}[[expr]]$, and $\mathcal{E}[[expr]]$ are as defined in [Wad99b]. The individual items are referred to below by IH1, ..., IH4. Since the expressions are variable-free, β is empty except handling the pseudo variables $Size$ and Pos (which are often also empty). □

Proof. The above theorem and lemma are proven by structural induction. The enumeration is the same as in Definition 5.9. Below, β is an assignment of the pseudo variables $Size$ and Pos (often even empty). We write $\stackrel{**}{=}$ for "equals by definition in [Wad99b]".

1. for closed, absolute expressions (i.e., without free variables),

   $$\mathcal{S}_{\mathcal{X}}(/expr) \stackrel{\text{Def}}{=} \mathcal{S}_{\mathcal{X}}(/expr, \emptyset) \stackrel{\text{IH1}}{=} \mathcal{S}[[/expr]](x)$$

   for arbitrary x.
2. reference expressions (only absolute expressions are considered)

   $$\mathcal{S}_{\mathcal{X}}(/p, \beta) \stackrel{\text{Def}}{=} \mathcal{S}^{any}_{\mathcal{X}}(p, root, \beta) \stackrel{\text{IH2}}{=} \mathcal{S}^{any}[[/expr]](root) \ .$$
3. location step:

   $$\mathcal{S}^{any}_{\mathcal{X}}(axis :: pattern, x, \beta) \stackrel{\text{Def}}{=} \mathcal{S}^{axis}_{\mathcal{X}}(pattern, x, \beta) \stackrel{\text{IH2}}{=} \mathcal{S}^{axis}[[/pattern]](x) \ .$$

4. the node test is the base case which is directly mapped to the axes:

$$\mathcal{S}^a_{\mathcal{X}}(name, x, \beta) \stackrel{\text{Def}}{=} \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,x)}(v \mid n = name)$$

which is characterized in [Wad99b] ($\mathcal{A}[[a]]$ enumerates the axes, $\mathcal{P}(a)$ gives the axes' principal nodetype) by

$$\{x_1 \mid x_1 \in \mathcal{A}[[a]]x,\ nodetype(x_1) = \mathcal{P}(a),\ \mathsf{name}(x_1) = name\}$$

which is the definition of $\mathcal{S}^a[[name]](x)$. Note that dereferencing `IDREF(S)` and splitting `NMTOKENS` has been excluded, thus, the result list is still in document order. Similar (note that node() is not defined in [Wad99b], we extend the definition according to the XPath specification)

$$\begin{array}{lcl} \mathcal{S}^a_{\mathcal{X}}(\mathsf{node}(), x, \beta) & \stackrel{\text{Def}}{=} & \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,x)}(v \mid v \in \mathcal{V}) \\ & = & \{x_1 \mid x_1 \in \mathcal{A}[[a]]x,\ nodetype(x_1) = element\} = \mathcal{S}^a[[\mathsf{node}()]](x) \\ \mathcal{S}^a_{\mathcal{X}}(\mathsf{node}(), x, \beta) & \stackrel{\text{Def}}{=} & \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,x)}(v \mid v \in \mathcal{V}) \\ & = & \{x_1 \mid x_1 \in \mathcal{A}[[a]]x,\ nodetype(x_1) = Text\} = \mathcal{S}^a[[\mathsf{text}()]](x)\ . \end{array}$$

5. step with variable: not allowed in the theorem.

6. filter(s):

$$\mathcal{S}^a_{\mathcal{X}}(pattern[filter], x, \beta) \stackrel{\text{Def}}{=} \mathsf{list}_{y\in\mathcal{S}^a_{\mathcal{X}}(pattern,x,\beta)}(y \mid \mathcal{Q}_{\mathcal{X}}(filter, y, \beta^{k,n}_{Pos,Size}))$$

where $L_1 \stackrel{\text{Def}}{=} \mathcal{S}^a_{\mathcal{X}}(pattern, x, \beta)$ which equals $\mathcal{S}^a[[pattern]](x, k, n)$ by induction hypothesis IH3 and $n := size(L_1)$, and for every y, let j the index of y in L_1 (which equals $size(\{x_1 \mid x_1 \in L_1, x_1 \leq_{doc} y\})$), $k := j$ if a is a forward axis, and $k := n{+}1{-}j$ if a is a backward axis. This is the same as defined for $\mathcal{S}^a[[pattern[filter]]](x)$.

This is – by induction hypothesis IH3 on $\mathcal{Q}$ the same as

$$\stackrel{\text{IH}}{=} \mathsf{list}_{y\in\mathcal{S}^a_{\mathcal{X}}(pattern,x,\beta)}(y \mid \mathcal{Q}[[filter]](,y,k,n)) \stackrel{**}{=} \mathcal{S}^a[[pattern[filter]]](x)\ .$$

7. Path:

$$\begin{array}{lcl} \mathcal{S}^a_{\mathcal{X}}(p_1/p_2, x, \beta) & \stackrel{\text{Def}}{=} & \mathsf{concat}_{y\in\mathcal{S}^a_{\mathcal{X}}(p_1,x,\beta)}(\mathcal{S}^{any}_{\mathcal{X}}(p_2, y, \beta)) \\ & \stackrel{\text{IH2}}{=} & \mathsf{concat}_{y\in\mathcal{S}^a[[p_1]](x)}(\mathcal{S}^a[[p_2]](y)) \stackrel{**}{=} \mathcal{S}^a[[p_1/p_2]](x)\ . \end{array}$$

Filters

8. Reference expressions (existential semantics) in filters:

$$\begin{array}{lcl} \mathcal{Q}_{\mathcal{X}}(refExpr, x, \beta) & \stackrel{\text{Def}}{\Leftrightarrow} & \mathcal{S}^{any}_{\mathcal{X}}(refExpr, x, \beta) \neq \emptyset \\ & \stackrel{\text{IH3}}{\Leftrightarrow} & \mathcal{S}^{child}[[refExpr]](x) \neq \emptyset \stackrel{**}{\Leftrightarrow} \mathcal{Q}[[refExpr]](x, k, n)\ . \end{array}$$

(for all k, n since these are not used in $refExpr$).

9. Predicate expressions: [Wad99b] knows only the "=" predicate; The definition is although not complete: e.g. for filters of the form [a/b/c = "foo"] which are allowed in XPath, there is no semantics defined. We extend the semantics according to the XPath specification, applying either $\mathcal{S}$ or $\mathcal{E}$.

$\mathcal{Q}_{\mathcal{X}}(pred(expr_1, \ldots, expr_n), x, \beta)$

$\stackrel{\text{Def}}{\Leftrightarrow}$ there are $x_1 \in \mathcal{S}^{any}_{\mathcal{X}}(expr_1, x, \beta), \ldots, x_n \in \mathcal{S}^{any}_{\mathcal{X}}(expr_n, x, \beta)$ such that $(x_1, \ldots, x_n) \in \mathcal{I}_P(pred)$

$\stackrel{\text{IH2/4}}{\Leftrightarrow}$ there are $x_1 \in \mathcal{S}^{child}[[expr_1]](x)$ or $x_1 \in \mathcal{E}[[expr_1]](x, \beta(Pos), \beta(Size)), \ldots,$ $x_n \in \mathcal{S}^{child}[[expr_n]](x)$ or $x_n \in \mathcal{E}[[expr_n]](x, \beta(Pos), \beta(Size))$

such that $pred(x_1, \ldots, x_n)$ holds.

10. Boolean Connectives

$$\begin{array}{lcl} \mathcal{Q}_{\mathcal{X}}(A \text{ and } B, x, \beta) & \overset{\text{Def}}{\Leftrightarrow} & \mathcal{Q}_{\mathcal{X}}(A, x, \beta) \text{ and } \mathcal{Q}_{\mathcal{X}}(B, y, \beta) \\ & \overset{\text{IH2}}{\Leftrightarrow} & \mathcal{Q}[[A]](x,k,n) \text{ and } \mathcal{Q}[[B]](x,k,n) \overset{**}{\Leftrightarrow} \mathcal{Q}[[A \text{ and } B]](x,k,n) \ . \end{array}$$

Analogously for "or" and "not".

11. Quantification is not allowed (since variables are not allowed at all)

12. Comparisons see predicates.

13. constants (in free predicates or in comparisons in filters); let *value* be a string or a number:

$$\mathcal{S}^{a}_{\mathcal{X}}(value, x, \beta) \overset{\text{Def}}{=} value \overset{**}{=} \mathcal{E}[[value]](x,k,n) \ .$$

14. variables are not allowed.

15. functions are not defined in [Wad99b], but the extension is obvious.

16. context-related functions use the extension of variable bindings by pseudo-variables *Size* and *Pos* in rule (6):

$$\begin{array}{lcccl} \mathcal{S}^{any}_{\mathcal{X}}(\mathsf{position}(), x, \beta) & \overset{\text{Def}}{\Leftrightarrow} & \beta(Pos) & \overset{**}{=} & \mathcal{E}[[\mathsf{position}()]](x, \beta(Pos), \beta(Size)) \\ \mathcal{S}^{any}_{\mathcal{X}}(\mathsf{last}(), x, \beta) & \overset{\text{Def}}{\Leftrightarrow} & \beta(Size) & \overset{**}{=} & \mathcal{E}[[\mathsf{last}()]](x, \beta(Pos), \beta(Size)) \ . \end{array}$$

17. arithmetics is proven similar to predicates; again handling the incompleteness of [Wad99b] as above:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{S}^{any}_{\mathcal{X}}(expr_1 \ \mathsf{op} \ expr_2, x, \beta)} \\ \overset{\text{Def}}{=} & \mathsf{set}_{x_1 \in \mathcal{S}^{any}_{\mathcal{X}}(expr_1, x, \beta), x_2 \in \mathcal{S}^{any}_{\mathcal{X}}(expr_2, x, \beta)}(x_1 \ \mathsf{op} \ x_2) \\ \overset{\text{IH2/4}}{=} & \mathsf{set}_{\substack{x_1 \in \mathcal{S}^{child}[[expr_1]](x) \text{ or } x_1 \in \mathcal{E}[[expr_1]](x, \beta(Pos), \beta(Size)), \ldots, \\ x_2 \in \mathcal{S}^{child}[[expr_2]](x) \text{ or } x_2 \in \mathcal{E}[[expr_2]](x, \beta(Pos), \beta(Size))}} (x_1 \ \mathsf{op} \ x_2) \\ \overset{**}{=} & \mathcal{E}[[expr_1 \ \mathsf{op} \ expr_2]](x, \beta(Pos), \beta(Size)) \ . \end{array}$$

18. Alternative/Union: not allowed in the theorem. □

Note that the above theorem does not extend to general closed formulas since join variables can further restrict the result set:

Corollary 5.4 (Correctness of $\mathcal{S}$ and $\mathcal{Q}$ wrt. XPath: Join Variables)
For closed absolute reference expressions (possibly containing bound variables) without navigating along reference attributes and splitting `NMTOKENS` attributes,

$$\mathcal{S}_{\mathcal{X}}(expr) \subseteq \mathcal{S}[[expr']](x)$$

where $expr'$ is obtained from $expr$ by removing all variables (including predicates which contain variables). □

The above theorems excluded the following functionality for handling attributes which is not provided by XPath:

- navigation along reference attributes: we intentionally define the order of the result set for dereferencing steps in a different way: the result is not in document order, but in the same order as the referencing elements were[1].

[1] if desired, a **reorder-to-document-order** operator may be added.

- splitting `NMTOKENS`: `NMTOKENS` are considered as atomic in XPath, whereas they are split in XPathLog.

Note the interaction of variable bindings by "$\rightarrow$V" and filtering:

Proposition 5.5 (Filters without Proximity Position Predicates)
The variable binding and the filtering construct commute if the filter does *not* use *proximity position predicates*: In this case, the expressions

$$/\mathit{path}/\mathit{axis}{::}\mathit{nodetest}{\rightarrow}\mathit{var}[\mathit{filter}]) \quad \text{and} \quad /\mathit{path}/\mathit{axis}{::}\mathit{nodetest}[\mathit{filter}]{\rightarrow}\mathit{var}$$

(where *var* does not occur somewhere else in the expression) are equivalent. □

Proof. Assume to be *expr* one of the above expressions. In both cases, $\mathcal{Q}_{\mathcal{X}}(\mathit{expr}, \mathit{root}, \beta)$ if

- $\beta(\mathit{var}) \in \mathcal{S}_{\mathcal{X}}(\mathit{path}/\mathit{axis}{::}\mathit{nodetest}, \mathit{root}, \beta \setminus \{\mathit{var}\})$ and
- $\mathcal{Q}_{\mathcal{X}}(\mathit{filter}, \beta(\mathit{var}), \beta \setminus \{\mathit{var}\})$. □

Both constructs do *not* commute if proximity predicates are used. Here it is important to compute the intermediate result list first and to apply the proximity position predicate and *then* bind the information from the selected element to variables (similar to grouping):

Example 5.6 (Filters: Proximity Position Predicates)
The expressions

//country[position()=5 and name→N]→C ,
//country[position()=5][name→N]→C , and
//country[position()=5]→C[name→N]

evaluate to true if C is bound to the 5th country in the database and N is bound to its name. In contrast, in

//country→C[name→N][position()=5]

already $\mathcal{S}_{\mathcal{X}}(//\mathsf{country}{\rightarrow}\mathsf{C}, \beta)$ is only non-empty if $\beta(C)$ is a country. Then, $\mathcal{S}_{\mathcal{X}}(//\mathsf{country}{\rightarrow}\mathsf{C}, \beta) = \beta(C)$. N is then bound to the name, but the result list does not contain a 5th element, thus, the expression evaluates to false.

5.5.2 Semantics of Formulas

Definition 5.10 (Semantics of XPath-Logic Formulas)
Formulas are interpreted according to the usual first-order semantics

$$\models \ \subseteq \ (\text{X-structures} \times \text{Var\_Assignments} \times \text{Formulas})$$

8. Reference Expressions: The semantics of reference expressions corresponds to a *predicate* in first-order logic, defining a purely existential semantics:

$$(\mathcal{X}, \beta) \models \mathit{refExpr} \ :\Leftrightarrow \ (\mathcal{S}_{\mathcal{X}}(\mathit{refExpr}, \beta)) \neq \emptyset$$

9. Predicates:

$$(\mathcal{X}, \beta) \models \mathit{pred}(A_1, \ldots, A_n) \ :\Leftrightarrow$$
$$\text{there are } x_1 \in \mathcal{S}_{\mathcal{X}}(A_1, \beta), \ldots, x_n \in \mathcal{S}_{\mathcal{X}}(A_n, \beta) \text{ s.t. } (x_1, \ldots, x_n) \in \mathcal{I}_P(\mathit{pred})$$

10. Boolean connectives use the definition from first-order logic:

$$
\begin{array}{lll}
(\mathcal{X},\beta) \models \mathsf{true}\ , & & \\
(\mathcal{X},\beta) \models fml_1 \wedge fml_2 & :\Leftrightarrow & (\mathcal{X},\beta) \models fml_1 \text{ and } (\mathcal{X},\beta) \models fml_2\ , \\
(\mathcal{X},\beta) \models fml_1 \vee fml_2 & :\Leftrightarrow & (\mathcal{X},\beta) \models fml_1 \text{ or } (\mathcal{X},\beta) \models fml_2\ , \\
(\mathcal{X},\beta) \models \neg fml & :\Leftrightarrow & \text{not } (\mathcal{X},\beta) \models fml\ , \\
(\mathcal{X},\beta) \models \exists x : fml & :\Leftrightarrow & \text{there is a } d \in \mathcal{U} \text{ with } (\mathcal{X},\beta_x^d) \models fml\ , \\
(\mathcal{X},\beta) \models \forall x : fml & :\Leftrightarrow & \text{for all } d \in \mathcal{U},\ (\mathcal{X},\beta_x^d) \models fml\ .
\end{array}
$$
□

We write $A \rightsquigarrow B$ for the implication which is usual denoted by "$\rightarrow$", since "$\rightarrow$" is used for variable binding in reference expressions.

The above semantics definitions associates a *truth value semantics* with XPath-Logic formulas. The $\models$ relation

$$\mathcal{X} \models closed\_formula$$

can be used for expressing integrity constraints on XML documents (see Example 5.7) and even sets of documents, and for reasoning on X-structures. In contrast, when defining XPathLog as a *data manipulation language* in the next section, a completely different formalization of the semantics is given: there, as for Datalog queries, the *answer substitutions* for a formula containing free variables have to be computed. The above definition of $\mathcal{S}$, defining a result list for an XPath-Logic reference expression is then extended.

Example 5.7 (Integrity Constraints)
Application-specific integrity constraints on the MONDIAL *database can be expressed in XPath-Logic.*

Range restrictions: *The text contents of population elements and the value of area attributes must be a non-negative number:*

∀ X,Y: ((//population/text()→X or Y/@area→X) ⇝ X ≥ 0).

Longitude/latitude values must be in the intervals $]-180, 180]$ *and* $[-90, +90]$*, respectively:*

∀ X: ((//longitude/text()→X ⇝ -180<X≤180) and (//latitude/text()→X ⇝ -90≤X≤90)).

The sum of percentages of ethnic groups in a country is less than 100%:

∀ C: (//country→C ⇝ sum{N [C]; ethnicgroups/@percentage→N} ≤ 100).

Bidirectional relationships: *The membership of countries in organizations is represented bidirectionally:*

∀ C,O: (//country→C[@memberships→O] ↔
 ∃ T: //organization→O/members[@type→T and @country→C]).

Referential integrity: *The* country *attribute of* border *subelements of* country *elements must reference a country which is encompassed by the same continent:*

∀ C,C2: (//country/border[@country→C2] ⇝
 (//country→C2 and
 ∃ Cont: (C/encompassed/@continent→Cont and
 C2/encompassed/@continent→Cont))).

Other conditions: *The capital of a country must be a city of the country:*

∀ C,Cap: (//country→C[@capital→Cap] ⇝ C//city→Cap).

DTDs and XML Schema. Instead of specifying the document structure by a DTD or an XML Schema instance, a set of closed XPath-Logic expressions can be given.

Example 5.8 (Integrity Constraints: DTD)
For instance, the definition of country *elements in the* MONDIAL *DTD (cf. Appendix A) which declares the set of allowed and required properties and their cardinalities can be represented by the following formula:*

```
∀ C: ( //country→C ⇝
      ((C/M ⇝ (M = name or M = population or ... or M = city)) and
       count{N [C]; C[name→N]} = 1 and
              ⋮
       count{N [C]; C[population_growth→N]} ≤ 1 and
              ⋮
       count{N [C]; C[encompassed→N]} ≥ 1 and
       (C/@M ⇝ (M = car_code or M = area or M = capital or M = memberships)) and
       count{N [C]; C[@car_code→N]} = 1 and
       count{N [C]; C[@capital→N]} = 1 and
       (C[@car_code→CC] ⇝ count{X [CC]; //country→X[@car_code→CC]} = 1)))
```

The DTD does not specify target types of reference attributes. Such conditions have to be added manually, as shown above for referential integrity. Also the datatypes, e.g., strings and numeric types are not considered in DTDs. XML Schema definitions can be translated into XPath-Logic formulas in a similar way.

5.5.3 Aggregation

The basic syntax is extended with aggregation as known for SQL. An aggregation term has the form

agg{X[G_1,...,G_n]; *body*}

where *agg* is one of the usual aggregation operators min, max, count, and sum and *body* is the aggregation body (that is, a conjunction of literals), and [G_1,...,G_n] are the group variables (similar to SQL). If the aggregation body contains other variables than the grouping variables, these are local to the aggregation. The grouping variables may occur anywhere in the rule body. Like arithmetic expressions, aggregation terms may only occur in the built-in predicates "=", "<", ">", "<=", ">=". The term *agg*{X[G_1,...,G_n]; *body*} returns one value for every tuple of values for [G_1,...,G_n]: All variable bindings satisfying body are calculated, yielding bindings for X,G_1,...,G_n. Then, the X's are grouped by [G_1,...,G_n] and for every group, *agg* is calculated and returned. Aggregation may be nested.
The list of grouping variables [G_1,...,G_n] is optional and may be omitted, e.g.,

?- N = count{C; //country→C}.

yields the total number of country elements in the database.

Definition 5.11 (Aggregation)
The body of an aggregation term is an XPath-Logic formula which shares the variable assignment of $G_1, \ldots, G_n$ with the surrounding environment. The semantics of formulas is used for evaluating the aggregation body:

$$\mathcal{S}_{\mathcal{X}}^{any}(\mathit{agg}\{\mathsf{V}[\mathsf{G}_1,\ldots,\mathsf{G}_n];\ \mathit{body}\}, x, \beta) = (\mathit{agg}(\{v \mid (\mathcal{X}, (\beta|_{\{G_1,\ldots,G_n\}})_V^v) \models \mathit{body}\}))$$

which results in a unary list. $(\beta|_{\{G_1,\ldots,G_n\}})$ denotes the restriction of β to the grouping variables $\{G_1, \ldots, G_n\}$ which are communicated between the body and its environment. □

5.6 Annotated Literals

The XML data model distinguishes between elements (i.e., nodes of the form <*tagname*> ... </*tagname*>) and their PCDATA contents. Nevertheless, as already pointed out in Examples 3.3 and 3.16, in several situations, elements containing PCDATA contents are expected to act as numbers or strings:

Example 5.9 (Annotated Literals)
Consider the following excerpt of the MONDIAL *XML instance:*

<country car_code="CH" ...> <name>Switzerland</name>
<population>7207060</population>
⋮
</country>

The XPath queries

//country[population > 5000000]/name/text() and
//country[population/text() > 5000000]/name/text()

both are equivalent and return "Switzerland" in their result set. In the first query, the element *<population>7207060</population> is implicitly casted into its literal value.*

What happens here is not evident in XML/DTD environments. The situation has already been illustrated in Example 3.3 for XPath queries, and in Example 3.16 when dealing with XML Schema *complexTypes* which are based on *simpleTypes*. The *complexType* is derived from a simple type which in some sense induces a "subclass" or "subtype" relationship: it must be allowed to substitute an instance of the *complexType* (i.e., an *element* with PCDATA contents) for a literal value (i.e., a text child). The idea here is that an element with a PCDATA contents adds structure to a simple type by allowing attributes (and also subelements in case of mixed contents). Thus, PCDATA elements with attributes behave as *annotated literals*:

- in comparisons, arithmetics, or (optionally) in the output, the literal value is used,
- in navigation expressions, the element node is used, and
- in variable bindings, the variable is bound to the element, but it acts as described above when the value of the variable is used e.g. in a comparison.
- when occurring in the head of an XPathLog rule (cf. Section 7) the element is used.

The interpretation of annotated literals is added to the definition of comparison predicates and arithmetics:

Definition 5.12 (Semantics of annotated Literals)
$\mathcal{S}$ is extended to $\mathcal{S}^*$ by

$$\mathcal{S}^*_{\mathcal{X}}(expr, v, \beta) = \mathsf{list}_{x \in \mathcal{S}^{any}_{\mathcal{X}}(expr,v,\beta)}(x \text{ if } x \in \mathcal{L}, \mathsf{string}(x) \text{ otherwise})$$

where $\mathsf{string}(x)$ is the XPath string function which returns the *string-value* of a node.

- Comparisons:

$$\mathcal{Q}_{\mathcal{X}}(expr_1 \ \mathsf{op}\ expr_2, y, \beta) \ :\Leftrightarrow$$
$$\text{there are } x_1 \in \mathcal{S}^*_{\mathcal{X}}(expr_1, y, \beta) \text{ and } x_2 \in \mathcal{S}^*_{\mathcal{X}}(expr_2, y, \beta) \text{ such that } x_1 \ \mathsf{op}\ x_2$$

where $\mathsf{op} \in \{<, \leq, >, \geq, =, \neq\}$.

- arithmetics:

$$\mathcal{S}_{\mathcal{X}}^{any}(expr_1 \text{ op } expr_2, x, \beta) = \mathsf{set}_{x_1 \in \mathcal{S}_{\mathcal{X}}^{*}(expr_1,x,\beta), x_2 \in \mathcal{S}_{\mathcal{X}}^{*}(expr_2,x,\beta)}(x_1 \text{ op } x_2)$$ □

Example 5.10 (Annotated Literals: XML Schema)
Consider the following excerpt of the MONDIAL *XML Schema description which declares a local* population *datatype (as a* complexType, *i.e., for use as an element type) which is derived from a* simpleType, *i.e., it is a literal:*

```
<complexType name="city">
    <element name="population" minOccurs="0" maxOccurs="unbounded">
      <complexType base="integer" derivedBy="extension">
        <attribute name="year" type ="date" use="optional"/>
      </complexType>
    </element>
</complexType>

<city id="cty-Germany-Berlin" country="D">
    <name>Berlin</name>
      <population year="95">3472009</population>
</city>
```

In XPath-Logic, the above fragment is modeled by complex objects with a text property as represented by the X-structure

$\mathcal{E}$(*berlin*,name,1) = *berlin-name*, $\mathcal{E}$(*berlin*,population,2) = *berlin-pop*,
$\mathcal{E}$(*berlin-name*,text(),1) = "Berlin", $\mathcal{E}$(*berlin-pop*,text(),1) = 951580,
$\mathcal{A}$(*berlin-pop*,year) = 1995.

which is described by the following facts:

berlin[name→*name-berlin* and population→*pop-berlin-95*].
pop-berlin-95[@year→1995 and text()→3472009].
name-berlin[text()→"Berlin"].

The elements pop-berlin-95 *and* name-berlin *are then casted as literals when required by the above specification. The user can treat* country/population *like a numerical value which is additionally annotated with a* year *attribute:*

//city/population *is an annotated literal:*

```
?- /city[name="Berlin" and population→P].
P/3472009
?- /city[name="Berlin" and population→P[@year→Y]].
P/3472009    Y/1995
?- /city[name="Berlin"]/population[@year→1995] > 3000000.
true
```

Although, 3472009[@year→1995] *does* not *hold. Note that the filter* [name="Berlin"] *also uses the text value of the* **PCDATA** *element.*

This problem is again investigated in Example 10.4 when dealing with the class hierarchy induced by an XML Schema description.

6 XPATHLOG: THE HORN FRAGMENT OF XPATH-LOGIC

Similar to the case of Datalog which is the quantifier-free Horn fragment of predicate logic, XPathLog is a logic programming language based on XPath-Logic. The language is evaluated wrt. suitable Herbrand structures.

Definition 6.1 (XPathLog Atoms)
Atoms are the basic components of XPathLog rules:

- an *XPathLog atom* is either a predicate expression, or an XPath-Logic *reference expression* which does neither contain quantifiers nor disjunction in filters.
- an XPathLog atom is *definite* if it uses only the child, sibling, and attribute axes and the atom does not contain negation, disjunction, function applications, and *proximity position predicates* (i.e., does not use the position() and last() functions). These atoms are allowed in rule heads (see Section 7.2); the excluded features would cause ambiguities what update is intended.
- an *XPathLog literal* is an atom or a negated atom,
- an *XPathLog query* is a list ?- L_1, ..., L_n of literals (in general, containing free variables),
- an *XPathLog rule* is a formula of the form

  $$A_1, \ldots, A_k \leftarrow L_1, \ldots, L_n$$

  where L_i are literals and A_i are definite atoms. $L_1, \ldots, L_n$ is the *body* of the rule, evaluated as a conjunction. $A_1, \ldots, A_k$ is the *head* of the rule, which may contain free variables that must also occur free in the body.

  Note that in contrast to usual Logic Programming, we allow for lists of atoms in the rule head which are interpreted as conjunctions,
- As usual, don't care variables are denoted by _X etc. □

In the remainder of this section, the semantics of queries is investigated. Since the rule bodies are queries, the query semantics is then also used for defining the semantics of XPathLog programs.

6.1 Queries in XPathLog

Similar to Datalog, the evaluation of a query ?- L_1, ..., L_n results in a set of variable bindings (of the free variables of the query) to *ground terms* taken from a suitable *Herbrand universe*:

Definition 6.2 (XML Herbrand Universe)
The XML Herbrand universe consists of

XML partition:

- $\mathcal{N}$ contains the name symbols from Σ: $\Sigma_N \cup \Sigma_C$ (attribute and element names, and constant names) (later also class names), and
- $\mathcal{V}$ (node identifiers), and

Literals: $\mathcal{L}$ (literal values, i.e., numbers and strings). □

Function symbols do not act as term constructors, but only for applying them to nodes. Thus, the Herbrand universe does not contain terms generated by functions. The semantics of the built-in functions of XPath is directly encoded into the evaluation.

Usual logic programming frameworks would define the notion of an XML Herbrand Structure based on a Herbrand base consisting of ground atoms which is then used to define a T_P operator. In the XML case, this is not appropriate since (i) the derived axes are redundant, and (ii) the insertion of subelements would require large changes in the structure. Instead, a Herbrand model based on the DOM [DOM98] idea is defined, similar to Definition 5.7: A *DOM Herbrand structure* consists of the following:

- a relational portion for the interpretation of (non-built-in) predicates which is represented in the same way as for Datalog by ground atoms of the Herbrand base. Note that with this, user-defined predicates involving literals, *names*, and *node identifiers* are allowed.
- an XML portion which represents the basic axes of an XML instance similar to X-structures, using only elements of the Herbrand universe.

Definition 6.3 (Herbrand Base)
For a given set $\mathcal{N}$ of names, the *Herbrand base* $\mathcal{HB}$ is the set of all *ground atoms* of the form

$$p(u_1, \ldots, u_n)$$

where $p \in \Sigma_P$ is a predicate symbol, and $u_i \in \mathcal{V} \cup \mathcal{N} \cup \mathcal{L}$ are elements of the Herbrand universe. □

Definition 6.4 (DOM Herbrand Structure)
A *DOM Herbrand structure* $\mathcal{HD}$ over a given Herbrand universe consists of

- a set $\mathsf{preds}(\mathcal{HD})$ of predicate atoms from $\mathcal{HB}$, and
- a (partial) mapping which associates with every $x \in \mathcal{V}$ two lists of ground pairs
  - $\mathcal{A}_{\mathcal{HD}}(\mathsf{child}, x) \in (\mathcal{V} \times \mathcal{N}_{\mathcal{E}})^{\mathbb{N}}$ and
  - $\mathcal{A}_{\mathcal{HD}}(\mathsf{attribute}, x) \in ((\mathcal{V} \cup \mathcal{L}) \times \mathcal{N}_{\mathcal{A}})^{\mathbb{N}}$ □

Again, there is a natural 1:1-correspondence between XML instances and DOM Herbrand structures (cf. Definitions 5.5 and 5.7).

The DOM Herbrand structure contains only the basic facts about the XML tree. Recall that $\mathcal{A}_{\mathcal{X}}(axis, x)$ has been defined for the derived axes based on the child axis in Definition 5.7; this definition also applies to DOM Herbrand structures, using $\mathcal{A}_{\mathcal{HD}}(\mathsf{child}, x)$.

Remark 6.1
Note that this approach, which associates the order with the children of elements differs from e.g., the DOM and XML-QL approaches where a *global* ordering of all elements is used. □

6.2 Semantics

The semantics of XPathLog queries which will be presented in this section associates a result set and a set of *answer substitutions* with every XPathLog query (in general containing free variables) by extending the definition of $\mathcal{S}$ given in Section 5.5. The semantics is based on the closed-world-assumption, i.e., all atoms which are assumed to hold in the model are explicitly stored in the database. The semantics also provides the formal base for the implementation of an *algebraic evaluation* of XPathLog queries in LoPiX (cf. Section 15).

6.2.1 Answers Data Model

The semantics of XPath-Logic reference expressions is defined wrt. an X-structure $\mathcal{X}$ as an *annotated result list*, i.e., every reference expression is mapped to a list of pairs where each pair consists of

(i) a result element, and

(ii) a set of variable bindings.

Formulas are mapped to a set of variable bindings.

Recall (see Remark 5.3) that the result sets of XPath expressions are *unordered*. Only internally to the computation, when single steps are considered, there is a *current node list* which is used for applying proximity position predicates.

Similar to Section 5.5, our semantics is again *list-based* – using the fact that *lists* are a specialization of *(multi)sets*.

Definition 6.5 (Semantics)
For a given set $V_1, \ldots, V_n$, of variables, the set of variable bindings is given as

$$((\mathcal{V} \cup \mathcal{L} \cup \mathcal{N})^n)^{\{V_1,\ldots,V_n\}} .$$

The set of *sets of* variable bindings for $V_1, \ldots, V_n$ – i.e., the possible answer sets for a query whose free variables are $V_1, \ldots, V_n$ – is

$$\text{Var\_Bindings}_{V_1,\ldots,V_n} \;:=\; (2^{((\mathcal{V}\cup\mathcal{L}\cup\mathcal{N})^n)})^{\{V_1,\ldots,V_n\}} .$$

Thus, in the general case for a general set Var of variables where n is unknown,

$$\text{Var\_Bindings} \;:=\; \bigcup_{n\in\mathbb{N}_0} (2^{((\mathcal{V}\cup\mathcal{L}\cup\mathcal{N})^n)})^{(\mathsf{Var}^n)}$$

is the set of sets of variable assignments. We use β for denoting an individual variable binding, and $\xi \in$ Var_Bindings for denoting a set of variable bindings.

For an empty set of variables, $\{true\}$ is the only element in Var_Bindings (representing the set with an empty variable assignment); in contrast, $\emptyset$ means that there is no variable binding which satisfies a given requirement.

$$\text{AnnotatedResults} \;:=\; ((\mathcal{V} \cup \mathcal{L}) \times \text{Var\_Bindings})$$

is the set of annotated results (i.e., an annotated result is a pair (v, ξ) where v is a node or a literal and ξ is a set of variable bindings (for the set of variables occurring free in a certain formula)). □

Less formally, the semantics of an expression is

(i) a result list, and

(ii) with every element of the result list, a list of variable bindings is associated.

The result list (i) is the same as defined by $\mathcal{S}$ in Definition 5.9, closely related to the one defined for XPath in [Wad99b].

Example 6.1 (Semantics)
First, the semantics is illustrated by an example, showing the difference between the result set and the additional bindings. Let $\mathcal{X}$ be the XML structure given in Example 5.5, let

$$\begin{aligned} expr := {} & //\mathsf{organization}{\rightarrow}\mathsf{O}[\mathsf{member/@country}[\mathsf{@car\_code}{\rightarrow}\mathsf{C}\ \mathsf{and}\ \mathsf{name/text()}{\rightarrow}\mathsf{N}]] \\ & \qquad /\mathsf{abbrev/text()}{\rightarrow}\mathsf{A}. \end{aligned}$$

Then, the semantics of expr is the annotated result list

$$\begin{array}{l} \mathsf{list}((\text{“UN”}, \{\mathsf{O}/\mathit{un}, \mathsf{A}/\text{“UN”}, \mathsf{C}/\text{“AL”}, \mathsf{N}/\text{“Albania”}, \\ \qquad\qquad \mathsf{O}/\mathit{un}, \mathsf{A}/\text{“UN”}, \mathsf{C}/\text{“GR”}, \mathsf{N}/\text{“Greece”}, \\ \qquad\qquad\qquad\qquad \vdots \qquad\qquad\qquad \}), \\ \quad (\text{“EU”}, \{\mathsf{O}/\mathit{eu}, \mathsf{A}/\text{“EU”}, \mathsf{C}/\text{“D”}, \mathsf{N}/\text{“Germany”}, \\ \qquad\qquad \mathsf{O}/\mathit{eu}, \mathsf{A}/\text{“EU”}, \mathsf{C}/\text{“F”}, \mathsf{N}/\text{“France”}, \\ \qquad\qquad\qquad\qquad \vdots \qquad\qquad\qquad \}), \\ \qquad \vdots \qquad\qquad\qquad\qquad\qquad\qquad) \end{array}$$

Definition 6.6 (Operators on Annotated Result Lists)
First, operations are defined which are needed for evaluating reference atoms in XPathLog.

Accessors: From an annotated result list θ, the result list is obtained as $\mathrm{Res}(\theta)$:

$$\begin{array}{rl} \mathrm{Res}\ : & \mathrm{AnnotatedResults}^{\mathbb{N}} \rightarrow (\mathcal{V} \cup \mathcal{L} \cup \mathcal{N})^{\mathbb{N}} \\ & ((x_1, \xi_1), \ldots, (x_n, \xi_n)) \mapsto (x_1, \ldots, x_n) \end{array}$$

For an annotated result list θ and a given node-identifier/literal/name x contained in the result list, the set of variable bindings associated with x, is obtained by $\mathrm{Bdgs}(\theta, x)$:

$$\begin{array}{rl} \mathrm{Bdgs}: & \mathrm{AnnotatedResults}^{\mathbb{N}} \times (\mathcal{V} \cup \mathcal{L} \cup \mathcal{N}) \rightarrow \mathrm{Var\_Bindings} \\ & (((x_1, \xi_1), \ldots, (x, \xi), \ldots, (x_n, \xi_n)), x) \mapsto \xi \\ & (\text{set } \mathrm{Bdgs}(\theta, x) = \emptyset \text{ if } x \notin \mathrm{Res}(\theta)) \end{array}$$

□

6.2.2 Safety

The semantics definition evaluates formulas and expressions wrt. a given set of variable bindings which e.g., results from evaluating other subexpressions of the same query. This approach allows for a more efficient evaluation of joins (*sideways information passing strategy*), and is especially needed for evaluating *negated* expressions (by defining negation as a relational “minus” operator). Negated expressions which contain free variables are intended to *exclude* some bindings from a given set of potential results. Thus, for variables occurring in the scope of a negation, the input answer set to the negation must already provide potential bindings. This leads to a *safety* requirement similar to Datalog.

Definition 6.7 (Safe Queries)
First, safety of variables is decided for each individual ocurrence. A variable occurrence V which is not inside an aggregation, is *safe* wrt. the query if at least one of the following holds:

- if the occurrence is in a literal L, and it is not inside the scope of a negation and not in a comparison predicate other than equality (e.g., $X < 3$ is unsafe).
- if the occurrence is in a literal L_i inside a filter $pattern[L_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ L_n]$ and V has a safe occurrence in *pattern* or in some L_j s.t. $j < i$.
- if the occurrence is in a literal L_i of the query $?\!-\!L_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ L_n$, and V has a safe occurrence in some L_j s.t. $j < i$.

The bodies of aggregation terms must be safe for themselves:

- An occurrence of a variable V in the *body* of an aggregation expression is safe if it is safe wrt. the query ?- *body*.

A query ?- L_1, ..., L_n is *safe* if all variable occurrences in the query are safe. □

Note that many unsafe queries can be rewritten into safe queries by

- reordering literals of the query,
- reordering literals inside filters,
- splitting reference expressions using additional variables, e.g., //country/city/population→P is equivalent to //country→_C, _C/city→_Cty, _Cty/population→P. Then, the individual atoms may be reordered.

6.2.3 Semantics of Expressions

In the following, the evaluation of safe queries is defined. The definition below uses the prerequisite that reference expressions are safe when evaluating them and their filters from left-to-right in rule (12) by applying left-to-right propagation when evaluating filters. Then, all variable evaluations in rule (15) are safe.
The basic (non-annotated) result lists are again provided by $\mathcal{A}_{\mathcal{HD}}(axis, v)$ for every node v of $\mathcal{HD}$ (cf. Definition 5.7).

Definition 6.8 (Semantics of XPath-Logic expressions)
The semantics is defined by operators $\mathcal{SB}$ and $\mathcal{QB}$ derived from $\mathcal{S}$ and $\mathcal{Q}$ as defined in Definition 5.9; the $\mathcal{B}$ stands for the extension with variable bindings:

- $\mathcal{SB}_{\mathcal{HD}}$: (Reference_Expressions) $\rightarrow$ AnnotatedResults$^{\mathbb{N}}$
  (Reference_Expressions $\times$ Var_Bindings) $\rightarrow$ AnnotatedResults$^{\mathbb{N}}$
  (Axes $\times$ $\mathcal{V}$ $\times$ Reference_Expressions $\times$ Var_Bindings) $\rightarrow$ AnnotatedResults$^{\mathbb{N}}$

  evaluates reference expressions wrt. an axis, an (optional) context node and a given set of variable bindings and returns an annotated result list as described above. Arithmetic expressions are also reference expressions.

- $\mathcal{QB}_{\mathcal{HD}}$: (Predicate_Expressions $\times$ $\mathcal{V}$ $\times$ Var_Bindings) $\rightarrow$ Var_Bindings
  evaluates filter expressions wrt. a context node and returns a set of variable bindings.

Expressions are evaluated by $\mathcal{SB}$:

1. if no input bindings are given,

$$\mathcal{SB}_{\mathcal{HD}}(\mathit{refExpr}) = \mathcal{SB}_{\mathcal{HD}}(\mathit{refExpr}, \emptyset)$$

2. reference expressions are translated into location paths wrt. a start node:
   - entry points: rooted path

$$\mathcal{SB}_{\mathcal{HD}}(/p, Bdgs) = \mathcal{SB}^{any}_{\mathcal{HD}}(p, root, Bdgs)$$

   - entry points: constants $c \in \Sigma_C$ (which are also elements of the Herbrand universe):

$$\mathcal{SB}_{\mathcal{HD}}(c/p, Bdgs) = \mathcal{SB}^{any}_{\mathcal{HD}}(p, c, Bdgs)$$

- entry points: variables $V \in \mathsf{Var}$:

$$\mathcal{SB}_{\mathcal{HD}}(V/p, Bdgs) = \mathsf{concat}_{x \in \mathcal{A}_{\mathcal{HD}}(\mathsf{descendants}, root)\Downarrow_1}(\mathcal{SB}^{any}_{\mathcal{HD}}(p, x, Bdgs \bowtie \{V/x\}))$$

Remark: Here, the input bindings are used for optimization: if every $\beta \in Bdgs$ provides already bindings for the variable V, the *sideways information passing strategy* directly effects the join $\{V/x\} \bowtie Bdgs$, restricting the possible values for V which in fact results in

$$\mathcal{SB}_{\mathcal{HD}}(V/p, Bdgs) = \\ \mathsf{concat}_{\beta \in Bdgs, x=\beta(V), x \in \mathcal{A}_{\mathcal{HD}}(\mathsf{descendants}, root)\Downarrow_1}(\mathcal{SB}^{any}_{\mathcal{HD}}(p, x, Bdgs \bowtie \{V/x\}))$$

Thus, the propagation of bindings is not only necessary for handling negation but also provides a relevant optimization for positive literals.
Note that in the recursive call $\mathcal{SB}^{any}_{\mathcal{HD}}(p, x, Bdgs \bowtie \{V/x\})$, the propagated bindings are already augmented with the binding for V.

3. location step:

$$\mathcal{SB}^{any}_{\mathcal{HD}}(axis :: pattern, x, Bdgs) = \mathcal{SB}^{axis}_{\mathcal{HD}}(pattern, x, Bdgs)$$

where $pattern$ is of the form *nodetestremainder* where *remainder* is a sequence of filters and variable bindings. These are evaluated left to right, always applying the rightmost "operation" (filter or variable) to the result of the left part:

4. node test:

$$\begin{array}{lcl} \mathcal{SB}^{a}_{\mathcal{HD}}(name, x, Bdgs) & = & \mathsf{list}_{(v,n) \in \mathcal{A}_{\mathcal{HD}}(a,x),\ n=name}(v, \{true\} \bowtie Bdgs) \\ \mathcal{SB}^{a}_{\mathcal{HD}}(\mathsf{node}(), x, Bdgs) & = & \mathsf{list}_{(v,n) \in \mathcal{A}_{\mathcal{HD}}(a,x),\ v \in \mathcal{V}}(v, \{true\} \bowtie Bdgs) \\ \mathcal{SB}^{a}_{\mathcal{HD}}(\mathsf{text}(), x, Bdgs) & = & \mathsf{list}_{(v,n) \in \mathcal{A}_{\mathcal{HD}}(a,x),\ v \in \mathcal{L}}(v, \{true\} \bowtie Bdgs) \\ \mathcal{SB}^{a}_{\mathcal{HD}}(N, x, Bdgs) & = & \mathsf{list}_{(v,n) \in \mathcal{A}_{\mathcal{HD}}(a,x)}(v, \{N/v\} \bowtie Bdgs) \end{array}$$

5. step with variable:

$$\mathcal{SB}^{a}_{\mathcal{HD}}(pattern \to V, x, Bdgs) = \mathsf{list}_{(y,\xi) \in \mathcal{SB}^{a}_{\mathcal{HD}}(pattern, x, Bdgs)}(y, \xi \bowtie \{V/y\})$$

6. filter:

$$\mathcal{SB}^{a}_{\mathcal{HD}}(pattern[filter], x, Bdgs) = \\ \mathsf{list}_{(y,\xi) \in \mathcal{SB}^{a}_{\mathcal{HD}}(pattern, x, Bdgs),\ \mathcal{QB}_{\mathcal{HD}}(filter, y, \xi') \neq \emptyset}(y,\ \mathcal{QB}_{\mathcal{HD}}(filter, y, \xi') \setminus \{Pos, Size\})$$

If the filter does not contain *proximity position predicates*, $\xi' := \xi$, otherwise let $L := \mathcal{SB}^{a}_{\mathcal{HD}}(pattern, x, Bdgs)$, and then for every (y, ξ) in L, ξ' is obtained as follows, extending ξ with bindings of the pseudo variables $Size$ and Pos:

- start with $\xi' = \emptyset$,
- for every $\beta \in \xi$, the list $L' = \mathsf{list}_{(y,\xi) \in L \text{ s.t. } \beta \in \xi}(y)$ contains all nodes which are selected *for the variable assignment* β.
- Let now $size := size(L')$, and for every y, let j the index of x_1 in L', $pos := j$ if a is a forward axis, and $pos := size{+}1{-}j$ if a is a backward axis.
- add $\beta^{Size,Pos}_{size,pos}$ to ξ'.

After evaluating the filter, the pseudo variables $Size$ and Pos are again removed from the bindings.

7. path:

$$\mathcal{SB}^{a}_{\mathcal{HD}}(p_1/p_2, x, Bdgs) = \mathsf{concat}_{(y,\xi) \in \mathcal{SB}^{any}_{\mathcal{HD}}(p_1, x, Bdgs)} \mathcal{SB}^{a}_{\mathcal{HD}}(p_2, y, \xi)$$

Filters

8. Reference expressions (existential semantics) in filters:

$$\mathcal{QB}_{\mathcal{HD}}(\mathit{refExpr}, x, \mathit{Bdgs}) = \bigcup_{(y,\xi)\in\mathcal{SB}^{any}_{\mathcal{HD}}(\mathit{refExpr},x,\mathit{Bdgs})} \xi$$

9. Predicates except built-in comparisons:

$$\mathcal{QB}_{\mathcal{HD}}(\mathit{pred}(\mathit{arg}_1,\ldots,\mathit{arg}_n), x, \mathit{Bdgs}) = \bigcup_{\substack{(x_1,\xi_1)\in\mathcal{SB}^{any}_{\mathcal{HD}}(\mathit{arg}_1,x,\mathit{Bdgs}),\ldots,\\ (x_n,\xi_n)\in\mathcal{SB}^{any}_{\mathcal{HD}}(\mathit{arg}_n,x,\mathit{Bdgs}),\\ \mathit{pred}(x_1,\ldots,x_n)\in\mathsf{preds}(\mathcal{HD})}} \xi_1 \bowtie \ldots \bowtie \xi_n$$

10. Negated expressions which do *not contain any free variable*:

$$\mathcal{QB}_{\mathcal{HD}}(\mathsf{not}\ A, x, \mathit{Bdgs}) = \begin{cases} \mathit{Bdgs} & \text{if } \mathcal{QB}_{\mathcal{HD}}(A,x,\emptyset)=\emptyset, \\ \emptyset & \text{otherwise, i.e., if } \mathcal{QB}_{\mathcal{HD}}(A,x,\emptyset)=\{\mathit{true}\}. \end{cases}$$

11. For negated expressions which contain free variables, negation is interpreted as the "minus" operator (as known e.g., from the relational algebra) wrt. the given input bindings. Thus, all variables which occur free in A must be safe, i.e., every input variable binding has to provide a value for them.

    For two variable bindings β_1, β_2, let $\beta_1 \leq \beta_2$ if β_1 is subsumed by β_2 (i.e., all variable bindings in β_1 occur also in β_2). Intuitively, in this case, if β_1 is "abandoned", β_2 should also be abandoned.

$$\begin{aligned}&\mathcal{QB}_{\mathcal{HD}}(\mathsf{not}\ \mathit{expr}, x, \mathit{Bdgs}) = \\ &\qquad \mathit{Bdgs} - \{\beta \in \mathit{Bdgs} \mid \text{ there is a } \beta' \in \mathcal{QB}_{\mathcal{HD}}(\mathit{expr},x,\mathit{Bdgs}) \text{ s.t. } \beta \leq \beta'\}\end{aligned}$$

12. Conjunction (recall that disjunction is not allowed in XPathLog filters):

$$\begin{aligned}&\mathcal{QB}_{\mathcal{HD}}(\mathit{expr}_1 \ \mathsf{and}\ \mathit{expr}_2, x, \mathit{Bdgs}) = \\ &\qquad \mathcal{QB}_{\mathcal{HD}}(\mathit{expr}_1, x, \mathit{Bdgs}) \bowtie \mathcal{QB}_{\mathcal{HD}}(\mathit{expr}_2, x, \mathcal{QB}_{\mathcal{HD}}(\mathit{expr}_1, x, \mathit{Bdgs}))\end{aligned}$$

    Here, in case of negated conjuncts in the filter, the safety of variables has to be considered. The above definition assumes that by a left-to-right evaluation of conjuncts, the evaluation is safe (see Definition 6.7).

Comparisons

13. The built-in equality predicate "=" serves as an assignment if the left-hand side is a variable $V \in \mathsf{Var}$ which is *not* bound in Bdgs:

$$\mathcal{QB}_{\mathcal{HD}}(V = \mathit{expr}, x, \mathit{Bdgs}) = \bigcup_{(y,\xi)\in\mathcal{SB}^{any}_{\mathcal{HD}}(\mathit{expr},x,\mathit{Bdgs})} \xi \bowtie \{V/y\}$$

    All other built-in comparisons require all variables to be bound:

$$\mathcal{QB}_{\mathcal{HD}}(\mathit{expr}_1 \ \mathsf{op}\ \mathit{expr}_2, x, \mathit{Bdgs}) = \bigcup_{\substack{(x_1,\xi_1)\in\mathcal{SB}^{*}_{\mathcal{HD}}(\mathit{expr}_1,x,\mathit{Bdgs}),\ldots,\\ (x_2,\xi_2)\in\mathcal{SB}^{*}_{\mathcal{HD}}(\mathit{expr}_2,x,\mathit{Bdgs}),\\ x_1 \ \mathsf{op}\ x_2}} \xi_1 \bowtie \xi_2$$

    where $\mathcal{SB}^*$ is defined analogously to Definition 5.12 for handling annotated literals.

Evaluation of Terms

14. constants (in free predicates or in comparisons in filters); let *value* be a string or a number:

$$\mathcal{SB}^{any}_{\mathcal{HD}}(value, x, Bdgs) = (value, Bdgs)$$

For constants $c \in \Sigma_C$:

$$\mathcal{SB}^{any}_{\mathcal{HD}}(c, x, Bdgs) = (\mathcal{I}_C(c), Bdgs)$$

15. variables (in free predicates or in comparisons in filters); the variable occurrence must be safe:

$$\mathcal{SB}^{any}_{\mathcal{HD}}(var, x, Bdgs) = \mathsf{list}_{\beta \in Bdgs}(\beta(var), \beta)$$

16. functions:

$$\begin{aligned}&\mathcal{SB}^{any}_{\mathcal{HD}}(f(arg_1, \ldots, arg_n)), x, Bdgs) = \\ &\qquad \mathsf{list}_{\substack{(x_1,\xi_1) \in \mathcal{SB}^{any}_{\mathcal{HD}}(arg_1, x, Bdgs), \ldots, \\ (x_n,\xi_n) \in \mathcal{SB}^{any}_{\mathcal{HD}}(arg_n, x, Bdgs)}} (x.f(x_1, \ldots, x_n), \xi_1 \bowtie \ldots \bowtie \xi_n)\end{aligned}$$

where $x.f(x_1, \ldots, x_n)$ results from the built-in evaluation of f.

17. context-related functions use the extension of variable bindings by pseudo-variables $Size$ and Pos in rule (6):

$$\begin{aligned}&\mathcal{SB}^{any}_{\mathcal{HD}}(\mathsf{position}(), x, Bdgs) = \mathsf{list}_{\beta \in Bdgs}(\beta(Pos), \{\beta' \in Bdgs \mid \beta(Pos) = \beta'(Pos)\}) \\ &\mathcal{SB}^{any}_{\mathcal{HD}}(\mathsf{last}(), x, Bdgs) = \mathsf{list}_{\beta \in Bdgs}(\beta(Pos), \{\beta' \in Bdgs \mid \beta(Size) = \beta'(Size)\})\end{aligned}$$

Recall that the filter $\mathit{pattern}[i]$ where $i \in \mathbb{N}$ stands for $\mathit{pattern}[\mathsf{position}()=i]$.

18. arithmetics:

$$\mathcal{SB}^{any}_{\mathcal{HD}}(expr_1 \ \mathsf{op}\ expr_2, x, Bdgs) = \mathsf{list}_{\substack{(x_1,\xi_1) \in \mathcal{SB}^{*}_{\mathcal{HD}}(expr_1, x, Bdgs), \\ (x_2,\xi_2) \in \mathcal{SB}^{*}_{\mathcal{HD}}(expr_2, x, Bdgs), \\ \text{group by } x_1 \ \mathsf{op}\ x_2}} (x_1 \ \mathsf{op}\ x_2, \xi_1 \bowtie \xi_2)$$

19. aggregation:

$$\begin{aligned}&\mathcal{SB}^{any}_{\mathcal{HD}}(\mathit{agg}\{X[G_1, \ldots, G_n]; \mathit{body}\}, x, Bdgs) = \\ &\qquad \mathsf{list}_{\beta \in Bdgs}(\mathit{agg}(\{\beta'(X) \mid \beta' \in \mathcal{QB}(\mathit{body}, x, \beta|_{\{G_1, \ldots, G_n\}})\}), Bdgs)\end{aligned} \qquad \square$$

Note that aggregation does *not* change the input bindings: the body is evaluated completely independently, only using the bindings of the variables $G_1, \ldots, G_n$. If the aggregation body contains other variables than the grouping variables, these are local to the aggregation; their bindings are not communicated to the environment.

Since the aggregation body is itself evaluated as a nested query, it has to obey the safety restrictions, i.e., no variable may represent an infinite answer set.

The above semantics is an algebraic-style extension to the logical semantics of XPath-Logic expressions which has been defined in Section 5.9:

Theorem 6.1 (Correctness of $\mathcal{SB}$ and $\mathcal{QB}$)
For every (in general, containing free variables) XPathLog expression *expr*,

$$\mathrm{Res}(\mathcal{SB}_{\mathcal{HD}}(expr)) = \bigcup_{\beta \in (\mathcal{V} \cup \mathcal{L} \cup \mathcal{N})^{\mathsf{free}(expr)}} \mathcal{S}_{\mathcal{HD}}(expr, \beta)\ .$$

More detailed, for all $x \in \mathcal{V} \cup \mathcal{L} \cup \mathcal{N}$,

$$(x \in \mathrm{Res}(\mathcal{SB}_{\mathcal{HD}}(expr)) \text{ and } \beta \in \mathrm{Bdgs}(\mathcal{SB}_{\mathcal{HD}}(expr), x)) \ \Leftrightarrow\ x \in \mathcal{S}_{\mathcal{HD}}(expr, \beta)\ . \qquad \square$$

Again, the theorem uses a lemma which encapsulates the structural induction.

Lemma 6.2 (Correctness of $\mathcal{SB}$ and $\mathcal{QB}$: Structural Induction)
The correctness of the answers semantics of XPathLog expressions mirrors the generation of answer sets by the evaluation: The input set *Bdgs* of Bindings *may* contain bindings for the free variables of an expression. If for some variable *var*, no binding is given, the result extends *Bdgs* with bindings of *var*. If bindings are given for *var*, this specifies a constraint on the answers to be returned (expressed by joins).

- For every absolute expression *expr*, (i.e., $expr = /expr'$) and every set *Bdgs* of variable bindings,

$$\begin{aligned}&(x \in \mathrm{Res}(\mathcal{SB}_{\mathcal{HD}}(expr, Bdgs)) \text{ and } \beta \in \mathrm{Bdgs}(\mathcal{SB}_{\mathcal{HD}}(expr, Bdgs), x)) \Leftrightarrow\\ &\qquad (x \in \mathcal{S}_{\mathcal{HD}}(expr, \beta) \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(expr)) \ .\end{aligned}$$

- For every expression *expr*, every node v, and every set *Bdgs* of variable bindings,

$$\begin{aligned}&(x \in \mathrm{Res}(\mathcal{SB}_{\mathcal{HD}}(expr, v, Bdgs)) \text{ and } \beta \in \mathrm{Bdgs}(\mathcal{SB}_{\mathcal{HD}}(expr, v, Bdgs), x)) \Leftrightarrow\\ &\qquad (x \in \mathcal{S}_{\mathcal{HD}}(expr, v, \beta) \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(expr)) \ .\end{aligned}$$

- for every filter *filter*, every node v, and every set *Bdgs* of variable bindings,

$$\begin{aligned}&\beta \in \mathcal{QB}_{\mathcal{HD}}(filter, v, Bdgs) \Leftrightarrow\\ &\qquad \mathcal{Q}_{\mathcal{HD}}(filter, v, \beta) \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(filter)) \ .\end{aligned}$$

□

Proof. The enumeration is the same as in the semantics definition. The proof is done by structural induction.

> **Note:** A bit sloppy, we write $(x, \beta) \in \mathcal{SB}_{\mathcal{HD}}(expr)$ for "$x \in \mathrm{Res}(\mathcal{SB}_{\mathcal{HD}}(expr))$ and $\beta \in \mathrm{Bdgs}(\mathcal{SB}_{\mathcal{HD}}(expr), x)$".

1. For closed expressions,

$$\begin{aligned}x \in \mathrm{Res}(\mathcal{SB}_{\mathcal{HD}}(refExpr)) &\overset{\mathrm{Def}}{\Leftrightarrow} x \in \mathrm{Res}(\mathcal{SB}_{\mathcal{HD}}(refExpr, \emptyset))\\ &\overset{\mathrm{IH}}{\Leftrightarrow} x \in \mathcal{S}_{\mathcal{HD}}(refExpr, \emptyset) \overset{\mathrm{Def.5.9}}{\Leftrightarrow} x \in \mathcal{S}_{\mathcal{HD}}(refExpr) \ .\end{aligned}$$

2. reference expressions are translated into location paths wrt. a start node:
   - entry points: rooted path

$$\begin{aligned}&(x, \beta) \in \mathcal{SB}_{\mathcal{HD}}(/p, Bdgs)\\ &\overset{\mathrm{Def}}{\Leftrightarrow} (x, \beta) \in \mathcal{SB}^{any}_{\mathcal{HD}}(p, root, Bdgs)\\ &\overset{\mathrm{IH}}{\Leftrightarrow} x \in \mathcal{S}^{any}_{\mathcal{HD}}(p, root, \beta) \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(/p))\\ &\overset{\mathrm{Def.5.9}}{\Leftrightarrow} x \in \mathcal{S}^{any}_{\mathcal{HD}}(/p, \beta) \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(/p)) \ .\end{aligned}$$

   - entry points: constants $c \in \mathcal{V}$ analogously (set c instead of *root* above)
   - entry points: variables $V \in \mathsf{Var}$. By definition,

$$\begin{aligned}&(x, \beta) \in \mathcal{SB}_{\mathcal{HD}}(V/p, Bdgs) \overset{\mathrm{Def}}{\Leftrightarrow}\\ &\qquad (x, \beta) \in \mathsf{concat}_{x \in \mathcal{A}_{\mathcal{HD}}(\mathsf{descendants}, root)\downarrow_1}(\mathcal{SB}^{any}_{\mathcal{HD}}(p, x, Bdgs \bowtie \{V/x\}))\end{aligned}$$

   which is exactly the case if there is an $x \in \mathcal{A}_{\mathcal{HD}}(\mathsf{descendants}, root)\downarrow_1$ such that

$$(x, \beta) \in (\mathcal{SB}^{any}_{\mathcal{HD}}(p, x, Bdgs \bowtie \{V/x\})) \ .$$

By induction hypothesis, this is equivalent with

$$x \in \mathcal{S}^{any}_{\mathcal{HD}}(p, x, \beta) \text{ and } \beta \text{ completes some } \beta' \in Bdgs \bowtie \{V/x\} \text{ with } \mathsf{free}(p)$$

which is exactly the case if $x = \beta(V)$ and β completes some $\beta' \in Bdgs$ with $\mathsf{free}(V/p)$. By Def. 5.9, this again is equivalent with

$$x \in \mathcal{S}^{any}_{\mathcal{HD}}(V/p, \beta) \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(V/p)\ .$$

3. location step:

$$\begin{array}{ll} (x,\beta) \in \mathcal{SB}^{any}_{\mathcal{HD}}(axis :: pattern, z, Bdgs) & \\ \overset{\text{Def}}{\Leftrightarrow} & (x,\beta) \in \mathcal{SB}^{axis}_{\mathcal{HD}}(pattern, z, Bdgs) \\ \overset{\text{IH}}{\Leftrightarrow} & x \in \mathcal{S}^{axis}_{\mathcal{HD}}(pattern, z, \beta) \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(pattern) \\ \overset{\text{Def.5.9}}{\Leftrightarrow} & x \in \mathcal{S}^{any}_{\mathcal{HD}}(axis :: pattern, z, \beta) \\ & \text{and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(axis :: pattern)\ . \end{array}$$

4. node test:

$$(x,\beta) \in \mathcal{SB}^{a}_{\mathcal{HD}}(name, z, Bdgs) \overset{\text{Def}}{\Leftrightarrow} (x,\beta) \in \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{HD}}(a,z),\ n=name}(v, \{true\} \bowtie Bdgs)$$

which is exactly the case if $x \in \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{HD}}(a,z),\ n=name}(v)$ and $\beta \in Bdgs$ which, by Def. 5.9 is equivalent with $x \in \mathcal{S}^{a}_{\mathcal{HD}}(name, z, \beta)$ and β completes some $\beta' \in Bdgs$ with $\mathsf{free}(name) = \emptyset$.

Analogously for $\mathsf{node}()$ and $\mathsf{text}()$.
Variables at nodetest position:

$$(x,\beta) \in \mathcal{SB}^{a}_{\mathcal{HD}}(N, z, Bdgs) \quad \overset{\text{Def}}{\Leftrightarrow} \quad (x,\beta) \in \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{HD}}(a,z)}(v, \{N/n\} \bowtie Bdgs)$$

which is exactly the case if $x \in \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{HD}}(a,z)}(v)$ and $\beta \in \{N/n\} \bowtie Bdgs$ which, by Def. 5.9 is equivalent with $x \in \mathcal{S}^{a}_{\mathcal{HD}}(N, z, \beta)$ and β completes some $\beta' \in Bdgs$ with $\mathsf{free}(N) = \{N\}$.

5. step with variable:

$$\begin{array}{ll} (x,\beta) \in \mathcal{SB}^{a}_{\mathcal{HD}}(pattern \to V, z, Bdgs) & \\ \overset{\text{Def}}{\Leftrightarrow} & (x,\beta) \in \mathsf{list}_{(y,\xi)\in\mathcal{SB}^{a}_{\mathcal{HD}}(pattern,z,Bdgs)}(y, \xi \bowtie \{V/y\}) \\ \Leftrightarrow & \text{there is a } \beta'' \text{ s.t. } (x,\beta'') \in \mathcal{SB}^{a}_{\mathcal{HD}}(pattern, z, Bdgs) \text{ and } \beta = \beta'' \bowtie \{V/x\}\ . \end{array}$$

By induction hypothesis, this is exactly the case if there is a β'' such that $x \in \mathcal{S}^{a}_{\mathcal{HD}}(pattern, z, \beta'')$ and β'' completes some $\beta' \in Bdgs$ with $\mathsf{free}(pattern)$, and $\beta = \beta'' \bowtie \{V/x\}$. Exactly then, since $x = \beta(V)$, by Def. 5.9, $x \in \mathcal{S}^{a}_{\mathcal{HD}}(pattern \to V, z, \beta)$ and β completes β' with $\mathsf{free}(pattern \to V) = \mathsf{free}(pattern) \cup \{V\}$.

6. filter(s):

$$\begin{array}{ll} (x,\beta) \in \mathcal{SB}^{a}_{\mathcal{HD}}(pattern[\mathsf{filter}], z, Bdgs) & \\ \overset{\text{Def}}{\Leftrightarrow} & (x,\beta) \in \mathsf{list}_{\substack{(y,\xi)\in\mathcal{SB}^{a}_{\mathcal{HD}}(pattern,z,Bdgs),\\ \mathcal{QB}_{\mathcal{HD}}(\mathsf{filter},y,\xi')\neq\emptyset}} (y,\ \mathcal{QB}_{\mathcal{HD}}(\mathsf{filter}, y, \xi') \setminus \{Pos, Size\}) \end{array}$$

for ξ as defined in Definition 6.8(6). This is exactly the case if

(a) there is a β'' s.t. $\beta'' \in \mathcal{QB}_{\mathcal{HD}}(\mathsf{filter}, x, \xi')$ and $\beta = \beta'' \setminus \{Pos, Size\}$,

(b) $(x,\xi) \in \mathcal{SB}^{a}_{\mathcal{HD}}(pattern, z, Bdgs)$ i.e., ξ is the corresponding set of variable bindings, and

(c) $\mathcal{QB}_{\mathcal{HD}}(\mathsf{filter}, x, \xi') \neq \emptyset$.

The first item is by induction hypothesis equivalent to

$$\mathcal{Q}_{\mathcal{HD}}(\mathit{filter}, x, \beta'') \text{ and } \beta'' \text{ completes some } \beta' \in \xi' \text{ with } \mathsf{free}(\mathit{filter}) \ . \qquad (*)$$

The third item is redundant here (it avoids the addition of elements with empty bindings list to the result). Since β'' completes some $\beta' \in \xi'$ with $\mathsf{free}(\mathit{filter})$, we know that $\gamma := \beta' \setminus \{Pos, Size\}$ is an element of ξ. Specializing the second item to γ yields

$$(x, \gamma) \in \mathcal{SB}^a_{\mathcal{HD}}(\mathit{pattern}, z, Bdgs) \ .$$

By induction hypothesis,

$$x \in \mathcal{S}^a_{\mathcal{HD}}(\mathit{pattern}, z, \gamma) \qquad (**)$$

and γ completes some $\gamma' \in Bdgs$ with $\mathsf{free}(\mathit{pattern})$. Above, we derived $\gamma = \beta' \setminus \{Pos, Size\}$. Using $(*)$, since β'' is a completion of β' with $\mathsf{free}(\mathit{filter})$, completing $\gamma' \in Bdgs$ first to γ (binding $\mathsf{free}(\mathit{pattern})$), then to β' (binding $Size$ and Pos), then to β'' (binding $\mathsf{free}(\mathit{filter})$), we have

$$\mathcal{Q}_{\mathcal{HD}}(\mathit{filter}, y, \beta'') \ .$$

From $(**)$, since β'' completes γ,

$$x \in \mathcal{S}^a_{\mathcal{HD}}(\mathit{pattern}, z, \beta'')$$

thus by Def. 5.9,

$$x \in \mathcal{S}^a_{\mathcal{HD}}(\mathit{pattern}[\mathit{filter}], z, Bdgs)$$

for β'' which completes $\gamma' \in Bdgs$ with $\mathsf{free}(\mathit{pattern}[\mathit{filter}])$.

The argumentation showed the "$\Rightarrow$" direction (which is the more difficult direction since γ must be guessed). "$\Leftarrow$" uses the same relationships and variable bindings.

7. Path:

$$\begin{aligned}
&(x, \beta) \in \mathcal{SB}^a_{\mathcal{HD}}(p_1/p_2, z, Bdgs) \\
&\quad \overset{\text{Def}}{\Leftrightarrow} \quad (x, \beta) \in \mathsf{concat}_{(y,\xi) \in \mathcal{SB}^{any}_{\mathcal{HD}}(p_1, z, Bdgs)} \mathcal{SB}^a_{\mathcal{HD}}(p_2, y, \xi) \\
&\quad \Leftrightarrow \quad \text{there is an } (y, \xi) \in \mathcal{SB}^{any}_{\mathcal{HD}}(p_1, z, Bdgs) \text{ s.t. } (x, \beta) \in \mathcal{SB}^a_{\mathcal{HD}}(p_2, y, \xi) \\
&\quad \overset{\text{IH}}{\Leftrightarrow} \quad \text{there is a } \gamma \in \xi \text{ s.t. there is a } \gamma' \text{ s.t. } x \in \mathcal{S}^a_{\mathcal{HD}}(p_2, y, \gamma') \text{ and} \\
&\qquad\qquad \gamma' \text{ completes } \gamma \text{ with } \mathsf{free}(p_2) \ .
\end{aligned}$$

For this γ, $(y, \gamma) \in \mathcal{SB}^{any}_{\mathcal{HD}}(p_1, z, Bdgs)$ and by induction hypothesis again $y \in \mathcal{S}^a_{\mathcal{HD}}(p_1, z, \gamma)$ and γ completes some $\beta' \in Bdgs$ with $\mathsf{free}(p_1)$. Thus, also $x \in \mathcal{S}^a_{\mathcal{HD}}(p_2, y, \gamma')$ and $y \in \mathcal{S}^a_{\mathcal{HD}}(p_1, z, \gamma')$ and by Def. 5.9, $x \in \mathcal{S}^a_{\mathcal{HD}}(p_1/p_2, z, \gamma')$. γ' completes some $\beta' \in Bdgs$ with $\mathsf{free}(p_1) \cup \mathsf{free}(p_2)$.

8. Reference expressions (existential semantics) in filters:

$$\begin{aligned}
&\beta \in \mathcal{QB}_{\mathcal{HD}}(\mathit{refExpr}, z, Bdgs) \\
&\quad \overset{\text{Def}}{\Leftrightarrow} \quad \beta \in \textstyle\bigcup_{(y,\xi) \in \mathcal{SB}^{any}_{\mathcal{HD}}(\mathit{refExpr}, z, Bdgs)} \xi \\
&\quad \Leftrightarrow \quad \text{there is a } y \text{ s.t. } (y, \beta) \in \mathcal{SB}^{any}_{\mathcal{HD}}(\mathit{refExpr}, z, Bdgs) \\
&\quad \overset{\text{IH}}{\Leftrightarrow} \quad y \in \mathcal{S}^{any}_{\mathcal{HD}}(\mathit{refExpr}, z, \beta) \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(\mathit{refExpr}) \\
&\quad \overset{\text{Def. 5.9}}{\Leftrightarrow} \quad \mathcal{Q}_{\mathcal{HD}}(\mathit{refExpr}, z, \beta) \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(\mathit{refExpr}) \ .
\end{aligned}$$

9. Predicates except built-in comparisons:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\beta \in \mathcal{QB}_{\mathcal{HD}}(pred(arg_1,\ldots,arg_n),z,Bdgs)} \\
\overset{\text{Def}}{\Leftrightarrow} & \beta \in \bigcup_{\substack{(x_1,\xi_1)\in\mathcal{SB}^{any}_{\mathcal{HD}}(arg_1,z,Bdgs),\ldots,\\ (x_n,\xi_n)\in\mathcal{SB}^{any}_{\mathcal{HD}}(arg_n,z,Bdgs),\\ pred(x_1,\ldots,x_n)\in\mathsf{preds}(\mathcal{HD})}} \xi_1 \bowtie \ldots \bowtie \xi_n \\
\Leftrightarrow & \text{there are } (x_1,\xi_1),\ldots,(x_n,\xi_n) \text{ s.t. } (x_i,\xi_i)\in\mathcal{SB}^{any}_{\mathcal{HD}}(arg_i,z,Bdgs) \\
& \text{and } pred(x_1,\ldots,x_n)\in\mathsf{preds}(\mathcal{HD}) \text{ and } \beta\in\xi_1\bowtie\ldots\bowtie\xi_n \\
\Leftrightarrow & (\text{take the right } \beta_i\in\xi_i) \\
& \text{there are } (x_1,\beta_1),\ldots,(x_n,\beta_n) \text{ s.t. } (x_i,\beta_i)\in\mathcal{SB}^{any}_{\mathcal{HD}}(arg_i,z,Bdgs) \\
& \text{and } pred(x_1,\ldots,x_n)\in\mathsf{preds}(\mathcal{HD}) \text{ and } \beta=\beta_1\bowtie\ldots\bowtie\beta_n \\
\overset{\text{IH}}{\Leftrightarrow} & \text{there are } (x_1,\beta_1),\ldots,(x_n,\beta_n) \text{ s.t. } x_i\in\mathcal{S}^{any}_{\mathcal{HD}}(arg_i,z,\beta_i) \\
& \text{and } \beta_i \text{ extends some } \beta'_i\in Bdgs \text{ with } \mathsf{free}(arg_i) \\
& \text{and } pred(x_1,\ldots,x_n)\in\mathsf{preds}(\mathcal{HD}) \text{ and } \beta=\beta_1\bowtie\ldots\bowtie\beta_n \\
\Leftrightarrow & (\text{the join guarantees that } \beta' := \beta'_1=\ldots=\beta'_n \text{ holds}) \\
& \text{there are } x_1,\ldots,x_n \text{ s.t. } x_i\in\mathcal{S}^{any}_{\mathcal{HD}}(arg_i,z,\beta_i) \\
& \text{and } \beta \text{ extends some } \beta'\in Bdgs \text{ with } \mathsf{free}(arg_1)\cup\ldots\cup\mathsf{free}(arg_n) \\
\overset{\text{Def.5.9}}{\Leftrightarrow} & \mathcal{Q}_{\mathcal{HD}}(pred(arg_1,\ldots,arg_n),z,\beta) \\
& \text{and } \beta \text{ completes some } \beta'\in Bdgs \text{ with } \mathsf{free}(pred(arg_1,\ldots,arg_n)) \ .
\end{array}
$$

10. Negated expressions which do *not contain any free variable*: trivial.

11. For negated expressions which contain free variables: Note that all variables in $\mathsf{free}(\mathsf{not}\ expr)$ are required to be bound by $Bdgs$ (safety).

$$
\begin{array}{ll}
\multicolumn{2}{l}{\beta \in \mathcal{QB}_{\mathcal{HD}}(\mathsf{not}\ expr,z,Bdgs)} \\
\overset{\text{Def}}{\Leftrightarrow} & \beta\in Bdgs \text{ and there is no } \beta'\in\mathcal{QB}_{\mathcal{HD}}(expr,z,Bdgs) \text{ s.t. } \beta\le\beta' \\
\overset{\text{IH}}{\Leftrightarrow} & \beta\in Bdgs \text{ and there is no } \beta'' \text{ such that} \\
& \mathcal{Q}_{\mathcal{HD}}(expr,z,\beta'') \text{ and } \beta'' \text{ extends } \beta' \text{ with } \mathsf{free}(expr) \text{ and } \beta\le\beta' \\
\overset{\text{Safety}}{\Leftrightarrow} & \beta\in Bdgs \text{ and not } \mathcal{Q}_{\mathcal{HD}}(expr,z,\beta) \\
\overset{\text{Def.5.9}}{\Leftrightarrow} & \beta\in Bdgs \text{ and } \mathcal{Q}_{\mathcal{HD}}(\ \mathsf{not}\ expr,z,\beta) \ .
\end{array}
$$

12. Conjunction (recall that disjunction is not allowed in XPathLog filters):

$$
\begin{array}{ll}
\multicolumn{2}{l}{\beta \in \mathcal{QB}_{\mathcal{HD}}(expr_1\ \mathsf{and}\ expr_2,z,Bdgs)} \\
\overset{\text{Def}}{\Leftrightarrow} & \beta\in\mathcal{QB}_{\mathcal{HD}}(expr_1,z,Bdgs)\bowtie\mathcal{QB}_{\mathcal{HD}}(expr_2,z,\mathcal{QB}_{\mathcal{HD}}(expr_1,z,Bdgs)) \\
\overset{\text{IH}}{\Leftrightarrow} & \text{there are } \gamma_1\in\mathcal{QB}_{\mathcal{HD}}(expr_1,z,Bdgs) \\
& \text{and } \gamma_2\in\mathcal{QB}_{\mathcal{HD}}(expr_1,z,\mathcal{QB}_{\mathcal{HD}}(expr_1,z,Bdgs)) \text{ s.t.} \\
& \mathcal{Q}_{\mathcal{HD}}(expr_1,z,\gamma_1) \text{ and } \gamma_1 \text{ completes some } \beta'\in Bdgs \text{ with } \mathsf{free}(expr_1) \text{ and} \\
& \mathcal{Q}_{\mathcal{HD}}(expr_2,z,\gamma_2) \text{ and } \gamma_1 \text{ completes some } \gamma''\in\mathcal{QB}_{\mathcal{HD}}(expr_1,z,Bdgs) \text{ with} \\
& \mathsf{free}(expr_2) \text{ and } \beta=\gamma_1\bowtie\gamma_2. \\
\Leftrightarrow & (\text{join condition: } \gamma_1=\gamma''\le\gamma_2) \\
& \mathcal{Q}_{\mathcal{HD}}(expr_1,z,\gamma_2) \text{ and } \mathcal{Q}_{\mathcal{HD}}(expr_2,z,\gamma_2) \\
& \text{and } \gamma_2 \text{ completes some } \beta'\in Bdgs \text{ with } \mathsf{free}(expr_1)\cup\mathsf{free}(expr_2) \\
\Leftrightarrow & \mathcal{Q}_{\mathcal{HD}}(expr_1\mathsf{and}\ expr_2,z,\gamma_2) \\
& \text{and } \gamma_2 \text{ completes some } \beta'\in Bdgs \text{ with } \mathsf{free}(expr_1)\cup\mathsf{free}(expr_2) \ .
\end{array}
$$

13. built-in equality predicate "=": similar to predicates and variable assigments by $\to V$. All other built-in comparisons: similar to predicates.

14. constants: trivial.

15. variables: trivial (safety!)

16. functions: similar to predicates

17. context-related functions use the extension of variable bindings by pseudo-variables $Size$ and Pos in rule (6):

$$\begin{array}{ll} \multicolumn{2}{l}{(x,\beta) \in \mathcal{SB}^{any}_{\mathcal{HD}}(\mathsf{position}(), z, Bdgs)} \\ \overset{\text{Def}}{\Leftrightarrow} & (x,\beta) \in \mathsf{list}_{\beta \in Bdgs}(\beta(Pos), \{\beta' \in Bdgs \mid \beta(Pos) = \beta'(Pos)\}) \\ \Leftrightarrow & \beta(Pos) = x \text{ for some } \beta \in Bdgs \\ \Leftrightarrow & x \in \mathcal{S}^{any}_{\mathcal{HD}}(\mathsf{position}(), z, \beta) \text{ for some } \beta \in Bdgs \\ \Leftrightarrow & x \in \mathcal{S}^{any}_{\mathcal{HD}}(\mathsf{position}(), z, \beta) \\ & \text{and } \beta \text{ completes some } \beta' \in Bdgs \text{ by } \mathsf{free}(\mathsf{position}()) \text{ (which is empty).} \end{array}$$

Analogously for $\mathsf{last}()$.

18. arithmetics: analogously to predicates.

19. aggregation: trivial (all free variables are required to be safe). □

6.3 Semantics of Queries

According to Definition 6.1, XPathLog queries are conjunctions of XPathLog literals. In the following, the evaluation of safe queries is defined. The definition of safety guarantees that a left-to-right evaluation of the body is well-defined (i.e., all variable evaluations in Definition 6.8(15) are safe). Definition 6.8(12) already applied left-to-right propagation when evaluating filters.

The evaluation $\mathcal{QB}$ defined in Definition 6.8 is extended to atoms by

$$\begin{array}{rl} \mathcal{QB}_{\mathcal{HD}} : & (\text{Atoms} \times \text{Var\_Bindings}) \to \text{Var\_Bindings} \\ & (A, Bdgs) \mapsto \mathcal{QB}_{\mathcal{HD}}(A, root, Bdgs) \end{array}$$

(instead of $root$, any $v \in \mathcal{V}$ can be used since A contains only absolute location paths.

Corollary 6.3 (Correctness: Evaluation of Atoms)
Queries consisting of only one atom can be evaluated without propagation, and $\mathcal{QB}$ returns the answer bindings:

- For every safe query ?- *refExpr* which consists of a single reference expression,

$$(\mathcal{HD}, \beta) \models refExpr \;:\Leftrightarrow\; \beta \in \mathcal{QB}_{\mathcal{HD}}(refExpr, \emptyset))\ .$$

- For every safe query ?- *pred(args)* which consists of a single predicate,

$$(\mathcal{HD}, \beta) \models pred(args) \;:\Leftrightarrow\; \beta \in \mathcal{QB}_{\mathcal{HD}}(pred(args), \emptyset))\ .$$

□

Definition 6.9 (Evaluation of Literals)
The evaluation of negated literals L is defined wrt. a set of input bindings which must cover the free variables in L, similar to negation in filters in Definition 6.8(11):

$$\mathcal{QB}_{\mathcal{HD}}(\mathsf{not}\ A, Bdgs) := Bdgs - \{\beta \in Bdgs \mid \text{ there is a } \beta' \in \mathcal{QB}_{\mathcal{HD}}(A, Bdgs) \text{ s.t. } \beta \leq \beta'\}\ .$$

□

Definition 6.10 (Evaluation of Queries)
The evaluation of a safe query

?- L_1, ..., L_n.

is defined similar to the evaluation of conjunctive filters in Definition 6.8(12):

$$\begin{aligned} \mathcal{QB}_{\mathcal{HD}} \ : \ & \text{Conj\_Literals} \rightarrow \text{Var\_Bindings} \\ & \mathcal{QB}_{\mathcal{HD}}(L_1 \wedge \ldots \wedge L_i) := \\ & \qquad \mathcal{QB}_{\mathcal{HD}}(L_1 \wedge \ldots \wedge L_{i-1}) \bowtie \mathcal{QB}_{\mathcal{HD}}(L_i, (\mathcal{QB}_{\mathcal{HD}}(true \wedge L_1 \wedge \ldots \wedge L_{i-1}))|_{\mathsf{free}(L_i)}) \end{aligned}$$

where $\mathcal{QB}_{\mathcal{HD}}(true) = \{true\}$, using left-to-right propagation. □

Theorem 6.4 (Correctness: Evaluation of Queries)
For all safe XPathLog queries Q,

$$\beta \in \mathcal{QB}_{\mathcal{HD}}(Q) \ \Leftrightarrow \ (\mathcal{HD}, \beta) \models Q \ .$$ □

Proof. Induction over the number of literals; base cases are provided by Lemma 6.2. □

Definition 6.11 (Answer set of a query)
Given a DOM Herbrand Structure $\mathcal{HD}$, the answer to a query

?- L_1, ..., L_n.

is the set

$$\mathsf{answers}_{\mathcal{HD}}(\mathsf{L}_1, \ldots, \mathsf{L}_n) \ := \ \mathcal{QB}_{\mathcal{HD}}(\mathsf{L}_1 \wedge \ldots \ \wedge \mathsf{L}_n)$$

of variable bindings. Recall that in the Herbrand-style context, the variables are bound to elements of the Herbrand universe. □

Note again that the semantics of formulas is not based on a Herbrand structure consisting of ground atoms (as "usual" Herbrand semantics are), but on the interpretations $\mathcal{A}_{\mathcal{HD}}$ of the axes in the DOM Herbrand structure.

7 XPATHLOG PROGRAMS

In logic programming, rules are used for a declarative specification: if the body of a clause evaluates to *true* for some assignment of its variables, the truth of the head atom for the same variable assignment can be inferred. Depending on the intention, this semantics can be used for (i) checking if something is derivable from a given set of facts, or (ii) extending a given set of facts by additional, derived knowledge. The second aspect is followed by *rule-based* approaches, which use rules for the specification of updates to a database or, generally, a situation.

Independent from the above, for logic programs, two evaluation strategies can be distinguished:

- Top-Down: refute or validate a claim, or compute an answer variable binding for a query.
- Bottom-up: collect all facts which can be derived from a given set of facts.

Note that bottom-up evaluation can also be applied for validating a claim or computing answers: the claim and the answers must be contained in the derived knowledge. Nevertheless, if only a single query is to be answered, the target-driven top-down evaluation which checks only the relevant parts of the input data is in general more efficient. On the other hand, from the database point of view, it is more common to derive all facts first (i.e., to extend the database), and then to answer questions.

In this work, we investigate the bottom-up strategy, regarding XPathLog as an *update language for XML databases*.

The body of an XPathLog rule is a set of XPathLog expressions. The evaluation of the body wrt. a given structure yields variable bindings which are propagated to the rule head where facts are added to the model.

7.1 Bottom-up Evaluation: Positive Programs

Positive XPathLog programs (i.e., the rules contain only positive literals; also filters may only contain positive expressions) are evaluated bottom-up by a T_P-like operator, providing a minimal model semantics. We first give the definition of T_P as defined for Prolog and Datalog:

Definition 7.1 (Deductive Fixpoint Semantics: The T_P-Operator)
For a Datalog program P and a Herbrand structure $\mathcal{H}$,

$$\begin{aligned}
T_P(\mathcal{H}) &:= \mathcal{H} \cup \{h \mid (h \leftarrow b_1, \ldots, b_n) \text{ is a ground instance of some rule of } P \\
&\qquad\qquad \text{and } b_i \in \mathcal{H} \text{ for all } i = 1, \ldots, n\} \ , \\
T_P^0(\mathcal{H}) &:= \mathcal{H} \ , \\
T_P^{i+1}(\mathcal{H}) &:= T_P(T_P^i(\mathcal{H})) \ , \\
T_P^\omega(\mathcal{H}) &:= \begin{cases} \lim_{i\to\infty} T_P^i(\mathcal{H}) & \text{if the sequence } T_P^0(\mathcal{H}), T_P^1(\mathcal{H}), \ldots \text{ converges,} \\ \perp \ \text{otherwise.} \end{cases}
\end{aligned}$$

The application of a Datalog program P to a given database $\mathcal{D}$ is defined as

$$P(\mathcal{D}) = T_P^\omega(\mathcal{D}) \ .$$

□

Preview. This definition is adapted in Definition 7.5 for XPathLog programs after explaining the semantics of insertions and updates. □

7.1.1 Atomization

In this section, another, *nearly* equivalent (it is equivalent for *definite XPathLog atoms* (cf. Definition 6.1), i.e., which do not contain negation, disjunction, quantifiers, and *proximity position predicates*; cf. Theorem 7.1) semantics of queries is defined which provides the base for the *constructive* semantics of reference expressions in *rule heads*. The semantics is defined by resolving reference expressions into their constituting atomic steps.

Remark 7.1
The counterpart of the resolving of expressions into atomic steps is followed by several approaches which store XML data in relational databases [DFS00, SGT+99, FK99], by flattening the XML instance to one or more universal relations and augmenting it by indexes. For these approaches, see Section 16.1. □

The function

$$\mathsf{atomize} : \mathrm{XPathLogAtoms} \rightarrow 2^{\mathrm{XPathLogAtoms}}$$

resolves a definite XPathLog atom into atoms of the form $\mathit{node}[\mathit{axis}::\mathit{nodetest}\rightarrow\mathit{result}]$ and predicates over variables and constants. It will be used in Definition 7.5 for specifying the semantics of rule heads. atomize is defined by structural induction corresponding to the induction steps when defining $\mathcal{S}_{\mathcal{X}}$:

Definition 7.2 (Atomization of Formulas)
The mapping

$$\mathsf{atomize} : \mathrm{XPathLogAtoms} \rightarrow 2^{\mathrm{XPathLogAtoms}}$$

resolves a definite XPath-Logic atom into a set of atomic expressions. In the following, *path* stands for a locationPath (or a variable), and *name* for a name (or a variable).

- the entry case:
$$\mathsf{atomize}(/remainder) := \mathsf{atomize}(\mathsf{root}/remainder)$$

- Paths are resolved into steps and filters are isolated (since proximity position predicates are not allowed in definite atoms, it can be assumed that there is at most one filter, optionally preceded by a variable assignment; cf. Proposition 5.5):

$$\begin{array}{l}\mathsf{atomize}(path/axis :: nodetest \rightarrow var[\mathit{filter}]\ /remainder) := \\ \quad \mathsf{atomize}(path[axis :: nodetest \rightarrow var]) \cup \mathsf{atomize}(var[\mathit{filter}]) \cup \mathsf{atomize}(var/remainder)\ , \\ \mathsf{atomize}(path/axis :: nodetest[\mathit{filter}]\ /remainder) := \\ \quad \mathsf{atomize}(path[axis :: nodetest \rightarrow \_X]) \cup \mathsf{atomize}(\_X[\mathit{filter}]) \cup \mathsf{atomize}(\_X/remainder) \\ \quad \text{where } \_X \text{ is a new don't care variable.}\end{array}$$

- Conjunctions in filters are separated:

$$\mathsf{atomize}(var[pred_1 \,\mathsf{and}\, \ldots \,\mathsf{and}\, pred_n]) := \mathsf{atomize}(var[pred_1]) \cup \ldots \cup \mathsf{atomize}(var[pred_n])$$

- Predicates in filters (including the case $var[expr]$; recall that quantifiers, negation, and disjunction are not allowed in definite XPath-Logic atoms):

$$\begin{array}{rl}\mathsf{atomize}(var[pred(expr_1, \ldots, expr_n)]) := & \mathsf{atomize}(equality(var, expr_1, \_X_1)) \cup \ldots \\ & \mathsf{atomize}(equality(var, expr_n, \_X_n)) \cup \\ & \{pred(\_X_1, \ldots, \_X_n)\}\end{array}$$

where $equality(var, expr, X)$ is defined as follows (if $expr_i$ is a constant, it is not replaced by a variable):

– $equality(var, expr, X) =$ "$expr \rightarrow X$" if $expr$ is of the form $//remainder$, or

– $equality(var, expr, X)$ = "$var/expr \rightarrow X$" if $expr$ is of the form $axis :: nodetest\ remainder$.

- Predicate atoms are handled in the same way. Note that here all arguments are absolute expressions (rooted, or starting at a constant). □

Example 7.1 (Atomization)

?- //organization→O[name/text()→ON and
@seat = members/@country[name/text()→CN]/@capital].

is atomized into

?- root[descendant::organization→O], O[name→_ON], _ON[text()→ON],
O[@seat→_S], O[members→_M], _M[@country→_C], _C[@country→_Cap], _S = _Cap,
_C[child::name→_CN], _CN[text()→CN].

Since for every definite XPathLog atom $expr$, atomize($expr$) is a valid XPathLog query, the semantics given in Definitions 5.9 and 6.8 also applies to atomized expressions.

Example 7.2 (Atomization)
The XML instance given in Example 5.1 verifies the following atoms (the extended syntax ***host[axis(i)::name→value]*** *is used in rule heads for specifying the position where to insert a new child or sibling, cf. Section 7.2).*

mondial[child(42)::country→*belgium*]	
belgium[@car_code→"B"]	*belgium*[@capital→*brussels*]
belgium[@memberships→*eu*]	*belgium*[@memberships→*nato*] ...
belgium[child(1)::name→*belgium-name*]	*belgium*[child(2)::name→*belgium-pop*]
belgium[child(3)::city→*brussels*]	*belgium*[child(4)::city→...]
belgium-name[child(1)::text()→"Belgium"]	*belgium-pop*[child(1)::text()→10170241]
brussels[@country→*belgium*]	*brussels*[@id→"city-brussels"]
brussels[child(1)::name→*brussels-name*]	*brussels*[child(2)::population→*brussels-pop*]
brussels-name[child(1)::text()→"Brussels"]	*brussels-pop*[child(1)::text()→951580]
mondial[child(45)::]country→*germany*]	
germany[@car_code→"D"]	*germany*[@capital→*berlin*]
germany[@memberships→*eu*]	*germany*[@memberships→*nato*] ...
mondial[child(179)::organization→*eu*]	
eu[@id→"org-eu"]	*eu*[@seat→*brussels*]
eu[child(1)::name]→*eu-name*]	*eu-name*[child(1)::text()→"European Union"]
eu]abbrev[child(2)::abbrev→*eu-abbrev*]	*eu-abbrev*[child(1)::text()→"EU"]
eu[child(3)::members→*eu-mem-mem*]	
eu-mem-mem[@type→"member"]	
eu-mem-mem[@country→*belgium*]	*eu-mem-mem*[@country→*belgium*] ...
eu[child(4)::members]→*eu-mem-appl*]	
eu-mem-appl[@type→"mem. appl."]	*eu-mem-appl*[@country→*albania*] ...
mondial[child(180)::organization→*nato*]	
nato[@id→"org-nato"]	*nato*[@seat→*brussels*]

(plus atoms for derived axes)

Theorem 7.1 (Correctness of atomize)
The above semantics is equivalent to the one presented in Definition 6.8 for all definite XPathLog

atoms A and every DOM Herbrand structure $\mathcal{HD}$, i.e.

$$\mathsf{answers}_{\mathcal{HD}}(A) = \mathsf{answers}_{\mathcal{HD}}(\mathsf{atomize}(A))$$

(recall that don't care variables are not considered in the answer.) □

Again, the theorem uses a lemma which encapsulates the structural induction. In Definitions 5.9 and 6.8, a logical and an algebraic semantics of XPath-Logic and XPathLog have been defined and shown to be equivalent in Theorem 6.1 and Lemma 6.2. Here, the logical semantics is used for showing the correctness of atomize.

Lemma 7.2 (Correctness of atomize: Structural Induction)
For every DOM Herbrand structure $\mathcal{HD}$ and every definite XPath-Logic atom A,

- for every variable assignment β of $\mathsf{free}(A)$ such that $(\mathcal{HD}, \beta) \models A$, there exists a variable assignment $\beta' \supseteq \beta$ of $\mathsf{free}(\mathsf{atomize}(A))$ such that $(\mathcal{HD}, \beta') \models \mathsf{atomize}(A)$, and
- for every variable assignment β' of $\mathsf{free}(\mathsf{atomize}(A))$ such that $(\mathcal{HD}, \beta') \models \mathsf{atomize}(A)$, $(\mathcal{HD}, \beta'|_{\mathsf{free}(A)}) \models A$. □

Proof. Structural induction.

- entry case (using $\beta = \beta'$):

$$\begin{aligned}
(\mathcal{HD}, \beta) \models /p \quad &\overset{\text{Def.5.10}}{\Leftrightarrow} \quad (\mathcal{S}_{\mathcal{HD}}(/p, \beta)) \neq \emptyset \overset{\text{Def.5.9}}{\Leftrightarrow} (\mathcal{S}_{\mathcal{HD}}(p, root, \beta)) \neq \emptyset \\
&\overset{\text{Def.5.9}}{\Leftrightarrow} \quad (\mathcal{S}_{\mathcal{HD}}(root/p, \beta)) \neq \emptyset \overset{\text{Def.5.10}}{\Leftrightarrow} (\mathcal{HD}, \beta) \models root/p \\
&\overset{\text{IH}}{\Leftrightarrow} \quad (\mathcal{HD}, \beta) \models \mathsf{atomize}(root/p) \\
&\overset{\text{Def}}{\Leftrightarrow} \quad (\mathcal{HD}, \beta) \models \mathsf{atomize}(/p) \ .
\end{aligned}$$

- Paths are resolved into steps and filters are isolated (the case where a don't care variable is introduced is shown; w.l.o.g., *path* is an absolute location path)

$$\begin{aligned}
&(\mathcal{HD}, \beta) \models path/axis :: nodetest[filter]\ /remainder \\
\Leftrightarrow\quad & \mathcal{S}_{\mathcal{HD}}(path/axis :: nodetest[filter]\ /remainder, \beta) \neq \emptyset \\
\Leftrightarrow\quad & \mathcal{S}_{\mathcal{HD}}(path/axis :: nodetest[filter]\ /remainder, root, \beta) \neq \emptyset \\
\Leftrightarrow\quad & \mathsf{concat}_{y \in \mathcal{S}^a_{\mathcal{HD}}(path/axis::nodetest[filter], root, \beta)}(\mathcal{S}^{any}_{\mathcal{HD}}(remainder, y, \beta)) \neq \emptyset \\
\Leftrightarrow\quad & \text{there is a node } v \in \mathcal{S}^a_{\mathcal{HD}}(path/axis :: nodetest[filter], root, \beta) \\
& \ \text{s.t. } \mathcal{S}^{any}_{\mathcal{HD}}(remainder, v, \beta) \neq \emptyset \\
\Leftrightarrow\quad & \text{there is a node } v \in \mathsf{list}_{y \in \mathcal{S}^a_{\mathcal{HD}}(path/axis::nodetest, x, \beta)}(y \mid \mathcal{Q}_{\mathcal{HD}}(filter, y, \beta)) \\
& \ \text{s.t. } \mathcal{S}^{any}_{\mathcal{HD}}(remainder, v, \beta) \neq \emptyset \\
\Leftrightarrow\quad & \text{there is a node } v \text{ s.t. } v \in \mathcal{S}^a_{\mathcal{HD}}(path/axis :: nodetest, x, \beta) \\
& \ \text{and } \mathcal{Q}_{\mathcal{HD}}(filter, v, \beta) \text{ and } \mathcal{S}^{any}_{\mathcal{HD}}(remainder, v, \beta) \neq \emptyset \\
\Leftrightarrow\quad & \text{there is a node } v \text{ s.t. } v \in \mathcal{S}^a_{\mathcal{HD}}(path/axis :: nodetest \to _X, x, \beta^v_{_X}) \\
& \ \text{and } \mathcal{Q}_{\mathcal{HD}}(V[filter], v, \beta^v_{_X}) \text{ and } \mathcal{S}^{any}_{\mathcal{HD}}(V/remainder, v, \beta^v_{_X}) \neq \emptyset \\
\Leftrightarrow\quad & \text{there is a node } v \text{ s.t. } v \in \mathcal{S}^a_{\mathcal{HD}}(path[axis :: nodetest \to _X], x, \beta^v_{_X}) \\
& \ \text{and } \mathcal{Q}_{\mathcal{HD}}(V[filter], v, \beta^v_{_X}) \text{ and } \mathcal{S}^{any}_{\mathcal{HD}}(V/remainder, v, \beta^v_{_X}) \neq \emptyset \\
\Leftrightarrow\quad & \text{there is a node } v \text{ s.t. } \mathcal{S}^a_{\mathcal{HD}}(path[axis :: nodetest \to _X], x, \beta^v_{_X}) \neq \emptyset \\
& \ \text{and } \mathcal{Q}_{\mathcal{HD}}(V[filter], v, \beta^v_{_X}) \text{ and } \mathcal{S}^{any}_{\mathcal{HD}}(V/remainder, x, \beta^v_{_X}) \neq \emptyset \\
\Leftrightarrow\quad & \text{there is a node } v \text{ s.t. } \mathcal{Q}_{\mathcal{HD}}(path[axis :: nodetest \to _X], \beta^v_{_X}) \\
& \ \text{and } \mathcal{Q}_{\mathcal{HD}}(V[filter], \beta^v_{_X}) \text{ and } \mathcal{Q}_{\mathcal{HD}}(V/remainder, \beta^v_{_X}) \\
\overset{\text{IH}}{\Leftrightarrow}\quad & \text{there is a node } v \text{ s.t. } \mathcal{Q}_{\mathcal{HD}}(\mathsf{atomize}(path[axis :: nodetest \to _X]), \beta^v_{_X}) \\
& \ \text{and } \mathcal{Q}_{\mathcal{HD}}(\mathsf{atomize}(V[filter]), \beta^v_{_X}) \text{ and } \mathcal{Q}_{\mathcal{HD}}(\mathsf{atomize}(V/remainder), \beta^v_{_X}) \\
\Leftrightarrow\quad & \text{there is a node } v \text{ s.t. } \mathcal{Q}_{\mathcal{HD}}(\mathsf{atomize}(\ldots), \beta^v_{_X}) \ .
\end{aligned}$$

- Conjunctions in filters: obvious.
- Predicates in filters: W.l.o.g., consider a unary predicate with a relative argument expression:

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\mathcal{HD},\beta) \models V[pred(expr)]} \\
\Leftrightarrow & \mathcal{S}_{\mathcal{HD}}(V[pred(expr)], \beta(V), \beta) \neq \emptyset \\
\Leftrightarrow & \mathsf{list}_{y \in \mathcal{S}^a_{\mathcal{HD}}(V,\beta(V),\beta)}(y \mid \mathcal{Q}_{\mathcal{HD}}(pred(expr), y, \beta)) \neq \emptyset \\
\Leftrightarrow & (\beta(V) \text{ is the only element in } \mathcal{S}^a_{\mathcal{HD}}(V, \beta(V), \beta)) \\
 & \mathcal{Q}_{\mathcal{HD}}(pred(expr), \beta(V), \beta) \\
\Leftrightarrow & \text{there is an } x \in \mathcal{S}_{\mathcal{HD}}(expr, \beta(V), \beta) \text{ such that } pred(x) \in \mathcal{HD} \\
\Leftrightarrow & \text{there is an } x \text{ s.t. } x \in \mathcal{S}_{\mathcal{HD}}(V/expr \to \_X, root, \beta^x_{\_X}) \text{ and } (\mathcal{HD}, \beta^x_{\_X}) \models pred(\_X) \\
\Leftrightarrow & \text{there is an } x \text{ s.t. } (\mathcal{HD}, \beta^x_{\_X}) \models V/expr \to \_X \text{ and } (\mathcal{HD}, \beta^x_{\_X}) \models pred(\_X) \\
\overset{\text{IH}}{\Leftrightarrow} & \text{there is an } x \text{ s.t. } (\mathcal{HD}, \beta^x_{\_X}) \models \mathsf{atomize}(V/expr \to \_X) \\
 & \text{and } (\mathcal{HD}, \beta^x_{\_X}) \models pred(\_X) \\
\Leftrightarrow & \text{there is an } x \text{ s.t. } (\mathcal{HD}, \beta^x_{\_X}) \models \mathsf{atomize}(V[pred(expr)]) \ .
\end{array}
$$

Predicate atoms: analogous. □

7.2 Left Hand Side

Since the right hand side of rules has already been handled in Section 6.1, the semantics of the left hand side is now investigated based on the atomization of expressions.

Using *logical expressions* for specifying an update is perhaps the most important difference to approaches like XSLT, XML-QL, or Quilt/XQuery where the structure to be *generated* is always specified by XML patterns (this implies that these languages do not allow for updating existing nodes[1] – e.g., adding children or attributes – but only for generating complete nodes). In contrast, in XPathLog, existing nodes are communicated via variables to the head, where they are *modified* when appearing at host position of atoms.

The head of an XPathLog rule is a set of *definite* XPathLog atoms. When used in the head, the "/" operator and the "[...]" construct specify which properties should be added or updated (thus, "[...]" does not act as a filter, but as a *constructor*). Recall that for the left hand side, proximity position predicates are not allowed; instead the position where a child or sibling should be inserted can be specified by

$$host[axis(i) :: name \to value]$$

where $axis$ is either child or a sibling axis (cf. Example 7.2 and Definition 7.4). If no position is specified, the new element is appended at the end of the axis.

Definition 7.3 (Enumerating Axes)
The notation $host[axis(i) :: name \to value]$ is defined as a "shortcut" for

$$host[axis :: *[i][\mathsf{name}() = name] \to value]$$

which states that $value$ is the ith subelement, and that it is a $name$-subelement. □

Note that the (pure) language does *not* allow to delete or replace existing elements or attributes[2] – modifications are always monotonic in the sense that existing "things" remain (although, children may be inserted between already existing children which makes the evaluation of proximity position predicates non-monotonic; similar to the evaluation of aggregration operators in related approaches).

[1] A proposal for extending XML querying languages based on variable bindings will be published in [TIHW01] (see Section 3.12).

[2] suitable extensions, e.g., atoms of the form delete(*element*,*property*,*value*) can be defined as extensions. In this case, the handling of dangling references must be defined. Such extensions which would turn XPathLog into a "classical" procedural language are not handled in this work.

Modification of Elements. When using the child or attribute axis for updates, the host of the expression gives the element to be updated or extended; when a sibling axis is used, effectively the parent of the host is extended with a new subelement.

Generation or Extension of Attributes. A ground atom of the form $n[@a{\rightarrow}v]$ specifies that the attribute $@a$ of the node n should be set or extended with v. If v is not a literal value but a node, a reference to v is stored.

Example 7.3 (Adding Attributes)
We add the data code to Switzerland, and make it a member of the European Union:

C[@datacode→"ch"], C[@memberships→O] :-
    //country→C[@car_code="CH"], //organization→O[abbrev/text()→"EU"].

results in

<country datacode="ch" car_code="CH" industry="machinery chemicals watches"
    memberships="org-efta org-un org-eu ..."> ... </country>

We discuss insertion of new subelements below, after showing how to create elements.

Creation of Elements. Elements can either be created as *free* elements by atoms of the form /*name*[...] (meaning "some element of type *name*" – in the rule head, this is interpreted to create an element which is not a subelement of any other element), or as subelements.

Example 7.4 (Creating Elements)
We create a new (free) country element with some properties (cf. Figures 7.1 and 7.2):

/country[@car_code→"BAV" and @capital→X and city→X and city→Y] :-
    //city→X[name/text()="Munich"], //city→Y[name/text()="Nurnberg"].

Note that the two city elements are linked *as subelements. This operation has no equivalent in the "classical" XML model: these elements are now children of* two *country elements. Thus, changing the elements effects both trees.* Linking *is a crucial feature for efficient restructuring and integration of data (see Section 11).*

Insertion of Subelements and Attributes. Already existing elements can be assigned as subelements to existing elements by using filter syntax in the rule head: A ground instantiated atom $n[\mathsf{child} :: s \rightarrow m]$ makes m a subelement of type s of n. In this case, in the Herbrand representation, m is *linked* as n/s at the end of the children list.

Example 7.5 (Inserting Subelements)
The following two rules are equivalent to the above ones:

/country[@car_code→"BAV"].
C[@capital→X and city→X and city→Y] :-
    //city→X[name/text()→"Munich"], //city→Y[name/text()→"Nurnberg"],
    //country→C[@car_code="BAV"].

Here, the first rule creates a free element, whereas the second rule uses the variable binding of C to this element for inserting subelements and attributes.

In the above case, the position of the new subelement is not specified. If the atom is of the form $h[\mathsf{child}(i){::}s{\rightarrow}v]$ or $h[\mathsf{following/preceding\text{-}sibling}(j){::}s{\rightarrow}v]$, this means that the new element to be inserted should be made the ith subelement of h or jth following/preceding sibling of h.

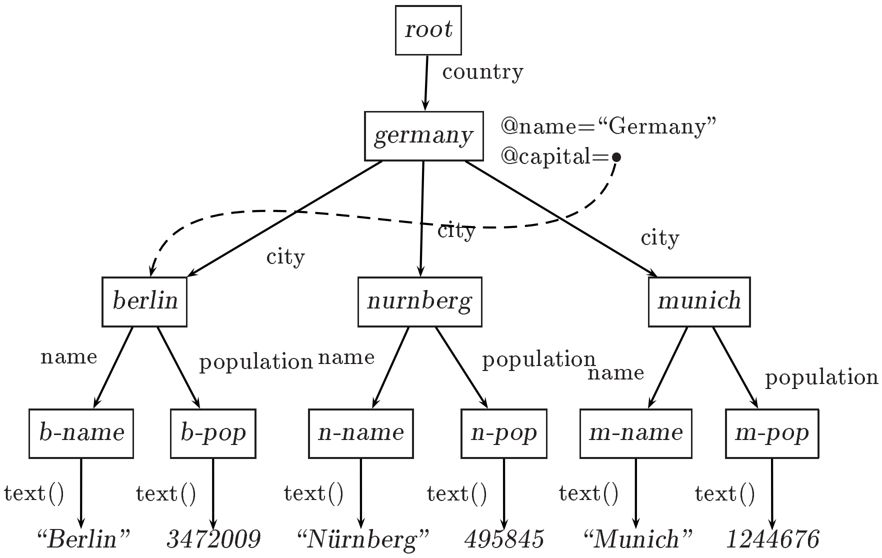

Figure 7.1: Linking – before

Once-for-each-Binding. In contrast to classical Logic Programming where it does not matter if a fact is "inserted" into the database several times (e.g., once in every round, conforming with the formal definition of the T_P operator), here subelements must be created exactly once for each instantiation of a rule (cf. Example 7.8). We consider this with defining a revised T_P-operator in Definition 7.5.

Generation of Elements by Path Expressions. Additionally, subelements can be created by *path expressions* in the rule head which create nested elements which satisfy the given path expression. The atomization introduces local variables which occur *only in the head* of the rule, i.e., the result is not a valid Logic Programming rule. Here, we follow the semantics of PathLog [FLU94] which is implemented in [LHL+98] for object creation. After the atomization, the resulting atoms are processed in an order such that the local variables are bound to the nodes/objects which are generated. Thus, the rules are in fact safe.

Example 7.6 (Inserting Text Children)
Bavaria gets a (PCDATA) subelement name:

C/name[text()→"Bavaria"] :- //country→C[@car_code="BAV"].

Here, the atomized version of the rule is

C[name→_N], _N[text()→"Bavaria"] :- root[descendant::country→C], C[@car_code="BAV"].

The body produces the variable binding C/bavaria. When the head is evaluated, first, the fact bavaria[child::name→x_1] is inserted, adding an (empty) name subelement x_1 to bavaria and binding the local variable _N to x_1. Then, the second atom is evaluated, generating the text contents to x_1.

Using Navigation Variables for Restructuring For data restructuring and integration, the intuitiveness and declarativeness of a language gains much from variables ranging not only over data, but also over schema concepts (classically, relations and columns, as, e.g., in SchemaSQL [LSS96b]). Such features have already been used for HTML-based Web data integration with F-Logic [KLW95, LHL+98].

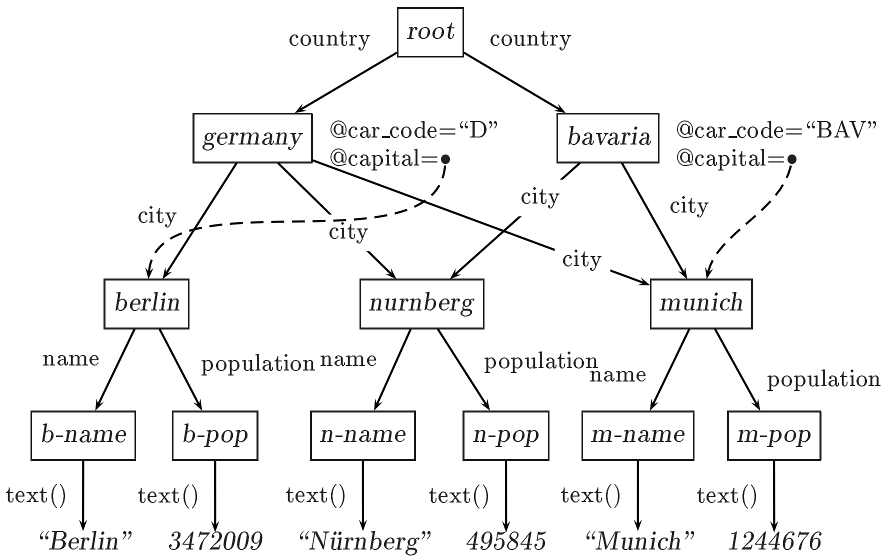

Figure 7.2: Linking – after

Extending the XPath wildcard concept, XPathLog also allows to have variables at name position. Thus, it allows for schema querying, and also for generating new structures dependent on the data contents of the original one. Here, the we define the semantics of XPathLog to cast strings into names when a variable is bound to a string in the body, and occurs at name position in the head:

Example 7.7 (Restructuring, Name Variables)
Consider another data source which provides data about waters according to the DTD

<!ELEMENT terra (water+, ...)>
<!ELEMENT water (...)> <!ATTLIST water name CDATA #REQUIRED ...>

which contains, e.g., the following elements:

<water type="river" name="Mississippi"> ... </water>
<water type="sea" name="North Sea"> ... </water> .

This tree should be converted into the target DTD

<!ELEMENT geo ((river|lake|sea)*)>
<!ELEMENT river (...)> <!ATTLIST river name CDATA #REQUIRED ...>
(analogously for lakes and seas)

The first rule,

result/T[@name→N] :- //water[@type→T and @name→N].

creates

<river name="Mississippi"/> and <sea name="North Sea"/> .

Attributes and contents are then transformed by separate rules which use @name *for identification. Properties are copied by using variables at element name and attribute name position:*

X[@A→V] :- //water[@type→T and @name→N and @A→V], //T→X[@name→N].
X[S→V] :- //water[@type→T and @name→N and S→V], //T→X[@name→N].

Then, result *is a valid* geo *element.*

7.3 Semantics of Positive XPathLog Programs

An XPathLog program is a declarative specification how to manipulate an XML database, starting with one or more input documents. The semantics of XPathLog programs is defined by bottom-up evaluation based on a T_P operator similar to Datalog. Thus, the semantics coincides with the usual understanding of a stepwise process.

For implementing the once-for-each-binding approach, the T_P operator has to be extended with bookkeeping about the instances of inserted rule heads. Additionally, the insertion of subelements adds some nonmonotonicity: adding an atom n[child(i)::e→v] (or n[following/preceding-sibling(i)::e→v]) adds a new subelement at the ith position (or as ith sibling), making the original ith child/sibling the i+1st etc. In case of multiple extensions to the same element, the indexes are evaluated wrt. the original structure. If several insertions effect the same position, the subelements are inserted according to the occurrence in the document order. Note that this does not *exchange* the i-th child/sibling (which would possibly produce dangling references).

Definition 7.4 (Extension of DOM Herbrand Structures)
Given a DOM Herbrand structure $\mathcal{HD}$ and a set $\mathcal{I}$ of atoms as obtained from atomize which are to be inserted, the new DOM Herbrand structure

$$\mathcal{HD}' = \mathcal{HD} \prec \mathcal{I}$$

is obtained from $\mathcal{HD}$ as follows:

- initialize
$$\begin{aligned}&\mathcal{A}_{\mathcal{HD}'}(\mathsf{child}, x) := \mathcal{A}_{\mathcal{HD}}(\mathsf{child}, x)\ ,\\ &\mathcal{A}_{\mathcal{HD}'}(\mathsf{attribute}, x) := \mathcal{A}_{\mathcal{HD}}(\mathsf{attribute}, x)\ ,\\ &\mathsf{preds}(\mathcal{HD}') := \mathsf{preds}(\mathcal{HD}) \cup \{p \mid p \in \mathcal{I} \text{ is a predicate atom}\}\end{aligned}$$
  for all node identifiers x.

- for all elements of $\mathcal{A}_{\mathcal{HD}}(\mathsf{child}, x)$,
$$\alpha(\mathcal{A}_{\mathcal{HD}}(\mathsf{child}, x)[i]) := \mathcal{A}_{\mathcal{HD}'}(\mathsf{child}, x)[i]$$
  (α maps the indexing from the old list to the new one).

- for all atoms $p[\mathsf{child}(i) :: e \to y] \in \mathcal{I}$, insert (y, e) into $\mathcal{A}_{\mathcal{HD}'}(\mathsf{child}, p)$ immediately after $\alpha(\mathcal{A}_{\mathcal{HD}}(\mathsf{child}, p)[i])$.

- for all atoms $p[\mathsf{child} :: e \to y] \in \mathcal{I}$, append (y, e) at the end of $\mathcal{A}_{\mathcal{HD}'}(\mathsf{child}, p)$.

- for all atoms $x[\mathsf{following\text{-}sibling}(i) :: e \to y] \in \mathcal{I}$ s.t. $p[\mathsf{child}(j) :: e' \to x] \in \mathcal{H}$, insert (y, e) into $\mathcal{A}_{\mathcal{HD}'}(\mathsf{child}, p)$ immediately after $\alpha(\mathcal{A}_{\mathcal{HD}}(\mathsf{child}, x)[j+i])$.

- for all atoms $x[\mathsf{preceding\text{-}sibling}(i) :: e \to y] \in \mathcal{I}$ s.t. $p[\mathsf{child}(j) :: e' \to x] \in \mathcal{H}$, insert (y, e) into $\mathcal{A}_{\mathcal{HD}'}(\mathsf{child}, p)$ immediately after $\alpha(\mathcal{A}_{\mathcal{HD}}(\mathsf{child}, x)[j–i])$.

- for all atoms $p[@a \to y] \in \mathcal{I}$, append (y, a) to $\mathcal{A}_{\mathcal{HD}'}(\mathsf{attribute}, p)$. □

Proposition 7.3 (Extension of DOM Herbrand Structures)
The extension operation is correct: $\mathcal{HD} \prec \mathcal{I} \models \mathcal{I}$, i.e., when querying the inserted atoms, the query evaluates to true. □

With the correctness of the atomize operation, the insertion of rule heads performs correctly:

Corollary 7.4 (Correctness of Insertions)
For inserting the ground-instantiated head of a rule, it is correct to insert the atomized head: For all ground XPathLog atoms A,

$$\mathcal{H}D \prec \mathsf{atomize}(A) \models A \ . \qquad \square$$

Definition 7.5 ($\mathcal{TX}_P$-Operator for XPath-Logic Programs)
The $\mathcal{TX}$-operator works on pairs $(\mathcal{HD}, Dic)$ where $\mathcal{HD}$ is a DOM Herbrand structure, and Dic is a dictionary which associates to every rule a set ξ of bindings which have been instantiated in the current iteration:

$$\begin{aligned}(\mathcal{HD}, Dic) + (\{(r_1,\beta_1),\ldots,(r_n,\beta_n)\}) \; &:= \; (\mathcal{HD} \prec \{\beta_i(\mathsf{atomize}(head(r_i))) \mid 1 \le i \le n\}, \\ &\qquad Dic.insert(\{(r_1,\beta_1),\ldots,(r_n,\beta_n)\})) \ , \\ (\mathcal{HD}, Dic)\downarrow_1 \; &:= \; \mathcal{HD} \ .\end{aligned}$$

For an XPathLog program P and a DOM Herbrand structure $\mathcal{HD}$,

$$\begin{aligned}\mathcal{TX}_P(\mathcal{HD}, B) &:= (\mathcal{HD}, B) + \{(r,\beta) \mid r = (h \leftarrow b) \in P \text{ and } \mathcal{HD} \models \beta(b), \text{ and } (r,\beta) \notin B\} \ , \\ \mathcal{TX}^0_P(\mathcal{HD}) &:= (\mathcal{HD}, \emptyset) \ , \\ \mathcal{TX}^{i+1}_P(\mathcal{HD}) &:= \mathcal{TX}_P(\mathcal{TX}^i_P(\mathcal{HD})) \ , \\ \mathcal{TX}^\omega_P(\mathcal{HD}) &:= \begin{cases} (\lim_{i\to\infty} \mathcal{TX}^i_P(\mathcal{HD}))\downarrow_1 & \text{if the sequence } \mathcal{TX}^0_P(\mathcal{HD}), \mathcal{TX}^1_P(\mathcal{HD}), \ldots \text{ converges,} \\ \perp \ \text{otherwise.} \end{cases}\end{aligned}$$

$\square$

The input to an XPathLog program is an XML document, respectively, the corresponding DOM Herbrand structure (recall that only the attribute axis and the child axis are stored which then unambiguously represent the tree structure).

Remark 7.2
Note that for pure Datalog programs P, evaluation wrt. $\mathcal{TX}_P$ does not change the semantics, i.e.,

$$\mathcal{TX}^\omega_P(\mathcal{HD}) = T^\omega_P(\mathcal{HD}) \ . \qquad \square$$

Proposition 7.5 (Properties of the $\mathcal{TX}_P$ operator)
The $\mathcal{TX}_P$ operator extends the well-known T_P operator. For all XPathLog programs P, the following holds:

- $\mathcal{TX}^\omega_P(\mathcal{HD}) \models P$,
- for positive XPathLog programs, and without considering proximity position predicates, the $\mathcal{TX}_P$ operator is monotonous, i.e.,
  - for all XPathLog queries A which do not use negation, aggregation, or proximity position predicates,
    $$\mathcal{HD} \models A \ \Rightarrow \ \mathcal{TX}_P(\mathcal{HD}) \models A \ ,$$
  - $\mathcal{TX}_P$ is order-preserving: for all XPathLog reference expressions $expr$ which do not use negation, aggregation, or proximity position predicates, $\mathcal{S}_{\mathcal{HD}}(expr)$ is a sublist of $\mathcal{S}_{\mathcal{TX}_P(\mathcal{HD})}(expr)$. $\square$

Proof. Both properties follow immediately from the definition. The child and attribute axes are extended solely by appending and inserting new "facts". $\square$

By its short, concise syntax, and exploiting the built-in semantics of the TX_P operator, XPathLog even allows for much shorter programs than, e.g., in XSLT for many tasks.

Example 7.8 (Inverting a Relationship)
Consider inverting the relation (country,language) *given by the CIA data source according to the DTD (cf. Appendix D)*

```
<!ELEMENT country (ethnicgroups*, religions*, languages*, ...)>

<!ELEMENT ethnicgroups (#PCDATA)>
<!ATTLIST ethnicgroups name CDATA #REQUIRED

<!ELEMENT religions (#PCDATA)>
<!ATTLIST religions name CDATA #REQUIRED>

<!ELEMENT languages (#PCDATA)>
<!ATTLIST languages name CDATA #REQUIRED>
```

to match the following DTD:

```
<!ELEMENT languageList (language+)>
<!ELEMENT language (spoken+)>    <!ATTLIST language name CDATA #REQUIRED>
<!ELEMENT spoken (EMPTY)>
    <!ATTLIST spoken country CDATA #REQUIRED    percent CDATA #REQUIRED>
```

Whereas in XSLT, the most common language for XML restructuring, this is a non-trivial task, it solely needs 2 1/2 rules in XPathLog:

```
//languageList→langs.
langs//language[@name→N] :- //country/languages[@name→N].
```

creates a language element for every language (note that this rule fires only once for each binding of the variable N) which is then filled by

```
L/spoken[@country→C and @percent→P] :-
    //country[@name→C]/languages[@name→N and text()→P],
    langs//language→L[@name→N].
```

and yields

```
<languageList>
    <language name='German'>
      <spoken country='Switzerland' percent='65'/>
      <spoken country='Germany' percent='100'/>
    </language>
    <language name='French'> ... </language>
    ⋮
</languageList>
```

7.4 Semantics of General XPathLog Programs

For logic programs which use negation (or similar nonmonotonic features, such as aggregation), there is no *minimal model semantics*. Instead, their semantics is defined wrt. perfect models, well-founded models, or stable models.

For practical use - especially when considering bottom-up evaluation – the notion of *perfect models* and *stratification* [Prz88] provides a solution to the problems raised by negation and other nonmonotonic features (such as e.g., aggregation). Stratification expresses the intuitive notion of process which executes as a sequence of steps:

> "... negation always refers to an already known relation. More specifically, first some relations should be defined (perhaps recursively) in terms of themselves *without* the use of negation. Next, some new relations can be defined in terms of themselves without the use of negation and in terms of the previous ones possibly with the use of negation. This process can be iterated."

Definition 7.6 (Stratification in Datalog)
A Datalog program P is stratified, if it can be partitioned into *strata* $P_1, \ldots, P_n$ such that for $i = 1, \ldots, n$,

- If a relation symbol R occurs *positively* in the body of a rule in P_i, then all rules containing R in their heads are contained in $\bigcup_{k \leq i} P_j$.
- If a relation symbol R occurs *negatively* in the body of a rule in P_i, then all rules containing R in their heads are contained in $\bigcup_{k < i} P_j$.

Then, P is *stratified via* $P_1 \cup \ldots \cup P_n$, and every P_i is a *stratum* of P. □

So far, the definition for Datalog. Note that not all Datalog programs are stratifiable – it is simply a syntactical criterion which is sometimes satisfied.

As long as variables are not allowed at the property position, a reasonable notion of stratification can be defined based on the names occurring at property position. With variables allowed at property position, it has been showed in [Fro98] that programs are in general not stratifiable. Since (i) even without variables at property position, there are many programs which are not syntactically stratifiable, and (ii) variables at the property position prove to be very useful for data integration (cf. Example 7.7), syntax-based stratification is not suitable for our approach.

Since the intention of XPathLog programs is in general to implement a stepwise process (which is also mirrored by the bottom-up evaluation strategy), often there is a natural, *user-defined* stratification. User defined stratification is supported in the LoPiX system [LoP] (cf. Section 15).

For stratified – including user-defined stratification – programs, the semantics is computed in the same way as for positive programs by iterating the $\mathcal{TX}_P$ operator.

8 A FIRST ANALYSIS OF XPATHLOG

The pure XPathLog language as presented up to now provides a data manipulation language for XML. As a first conclusion, XPathLog is compared with some XML querying and transformation languages and concepts which have been presented in the introduction.

8.1 Comparison with other XML Languages

8.1.1 XPathLog vs. Requirements

In [FSW99], XQL, XML-QL, and the languages YATL [CDSS99] and *Lorel* [AQM+97a, GMW99] are compared and essential features of an XML querying language have been identified. XPathLog relates to their requirements as follows:

- existence of some kind of pattern clause, filter clause, and constructor clause: pattern and filter clause are the same as in XPath, extended with variable bindings. The path patterns are superior to XML patterns (as e.g. used in XML-QL) since they allow for dereferencing (cf. page 37) and navigation along different axes. The constructor clause uses the same XPath-based syntax.
- constructs for imposing nesting and order: nested elements in the result tree are generated by subsequent rules which stepwise generate the result. Grouping (via stepwise generation) and order (via child(i)::name) is supported.
- use of a join operator to combine data from different sources: supported (for the handling of several sources, see Section 11).
- tag variables or path expressions: tag variables are supported, path expressions are not included in the basic XPathLog language (also not in XPath; any addition to XPath also translates to XPathLog). They are definable as derived relations.
- processing of alternatives: alternatives are expressible using a separate rule for each alternative.
- checking for absence of information: existence or non-existence of properties can be tested using negation, e.g. //country[not @indep_date].
- external functions: aggregation, string functions and some data conversion is built-in; the set of functions is extensible.
- navigation along references: implicit dereferencing is supported.

8.1.2 XQL

The ideas of XPath-Logic and XPathLog show some similarities with the XQL extensions [RLS98, Rob99] (cf. Section 3.3) to XSL Patterns/XPath: the use of join variables, similar to Prolog, Datalog, F-Logic, and similar languages, has been firstly applied in the XML world in XQL.

Since in XQL the querying expression also serves as generating expression for the result, the restructuring functionality is only limited. The distinction between querying (rule bodies) and generating (rule heads) in XPathLog by communicating variable bindings allows for a more flexible restructuring, including

- the generated result tree may contain element types on arbitrary tree levels which did not exist in the original tree,
- adding subelements and attributes to existing ones (update functionality).

8.1.3 XML-QL

XML-QL [DFF+99b] (see also Section 3.5) shares its declarative, "rule-based" nature with XPathLog. In some sense, using XML patterns in head (`CONSTRUCT`) and body (`WHERE`) clause, it is the XML-pattern-counterpart to the XPath-based XPathLog.

Non-Nested Queries. XML queries *without nesting* are comparable to individual XPathLog rules (without fixpoint evaluation): an XML query without nesting is actually a logic rule

CONSTRUCT *xml-pattern* :- WHERE *xml-pattern* IN *document*.

Both languages match an XML document against a pattern (XML patterns in XML-QL, and XPath patterns in XPathLog) for generating variable bindings. Then, the second part of the "rule" (the `CONSTRUCT` part in XML-QL, and the head in XPathLog) generates a result.

Nested Expressions. Nested expressions in the `WHERE - IN` clause have to be translated into rules which are executed *before* for creating intermediate result trees. If the nested query is not too complex, it is also often possible to rewrite it and incorporate it into the rule body.

Nested queries in the `CONSTRUCT` clause which serve for grouping are not directly translatable into the logic rule representing the main query. Instead, result grouping is done in XPathLog by having a set of rules which together create a complex tree. Here it is crucial that XPathLog is a *database programming language* instead of a pure querying or transformation language such as XQL and XML-QL: XPathLog rules

- update the underlying database instead of generating an isolated XML tree, and
- in this way, subsequent rules can be used to refine the results of previous ones.

Example 8.1 (Grouping in XML-QL and XPathLog)
Consider two data sources. Source 1 provides information about countries, source 2 provides information about cities (having an attribute country *= "name"). The result should be a document which nests city elements in country elements.*
The XML-QL query

```
WHERE <country name=$cname> </>
      IN source1
CONSTRUCT
      <country name=$cname>
      WHERE <city country = $cname name=$cityname> </>
            IN source2
      CONSTRUCT
            <city name=$cityname> </>
      </>
```

is equivalent to the XPathLog program

```
result/country[@name→CName] :- source1/country[@name→CName].
C/city[@name→Cityname] :- result/country→C[@name→CName],
                          source2/city[@country→CName and @name→Cityname].
```

where the first rule creates country elements and the second rule adds the city elements to the appropriate country elements.

The above example did not copy the complete element contents, but the result contained only the names of the country and the cities. A similar transformation, adding the whole element contents to the result can be specified in XPathLog as follows:

```
result/country→C :- source1/country→C.
C/city→City :- result/country→C[@name→CName],
               source2/city[@country→CName].
```

which simply links *the elements to the result tree. In contrast, XML-QL runs into problems since the country element should be copied completely,* and *it should be updated:*

```
WHERE <country name=$cname> </> ELEMENT_AS $country
        IN source1
CONSTRUCT
        <country name=$cname ?? >   <!– How to copy all attributes? –>
        WHERE <$a>$v</>in $country      <!– copy all subelements –>
        CONSTRUCT <$a>$v</>
        WHERE <city country = $cname>$c</> ELEMENT_AS $city
                IN source2
        CONSTRUCT
                $city
        </>
```

Here, it is not clear how to iterate over all attributes and add them to the generated element.

The generation of the result tree by *linking* elements as subelements to another element is important for data integration (see Section 11). Note that although the XML-QL data model presented in [DFF$^+$99b] as an edge-labled model would allow for linking, this is not supported by the XML-QL language (at least as long as XML-QL does not support updates at all; a potential combination of XML patterns and updates is not obvious).

In contrast to XPathLog, XML-QL does not support recursive queries. Recall also that the XML-QL patterns for selecting elements do not support the XML *axes* except the child axis, and indirectly the descendant by regular path expressions. On the other hand, regular path expressions are not yet supported in XPathLog (the syntax definition would allow an extension).

8.1.4 Quilt/XQuery.

XQuery (see Section 3.11) is the W3C's favorite XML querying language. The principal idea is the same both in XQuery and XPathLog (and also in SQL and OQL): generating variable bindings, and constructing results.

Translation from XQuery to XPathLog.

Non-Nested Queries. Similar as for XML-QL, Quilt/XQuery `FLWR` expressions without nesting can be regarded as logical rules

RETURN *xml-pattern* :- FOR ... LET ... WHERE ... .

Since XQuery also uses XPath expressions for binding variables, an even closer translation of the rule body can be given:

For an XQuery `FLW` statement (note that a mixed sequence of `FOR` and `LET` statements is allowed for binding variables to the result of other ones) where

- $x_1, \ldots, x_n$ are the variables defined by `FOR - LET` clauses (the definition of x_i may use $x_1, \ldots, x_{i-1}$),
- FOR var_{i_1} IN $xpath\text{-}expr_{i_1}, \ldots,$ FOR var_{i_k} IN $xpath\text{-}expr_{i_k}$ where $(i_1, \ldots, i_k)$ is a subsequence of $(1, \ldots, n)$ are the `FOR` clauses, and
- LET var_{j_1} IN $xpath\text{-}expr_{j_1}, \ldots,$ LET var_{j_l} IN $xpath\text{-}expr_{j_l}$ where $(j_1, \ldots, j_k)$ is a subsequence of $(1, \ldots, n)$ are the `LET` clauses
(thus, for every $x \in 1, \ldots, n$, x belongs either to the sequence $(i_1, \ldots, i_l)$ or to $(j_1, \ldots, j_k)$)
- without loss of generality, every var_{j_x} (i.e., variable which is bound by a `LET` statement) is bound to a result set containing more than one element - otherwise a `FOR` would be equivalent.
- $filter$ is the body of the `WHERE` clause,

the XPathLog rule body contains

- the atom "$xpath\text{-}expr_i \rightarrow var_i$" where var_i is bound by a `FOR` clause,
- the atom "$xpath\text{-}expr_i \rightarrow var_i$" where var_i is bound by a `LET` clause, and is used in an `IN` clause,
- the $filter$, where all aggregations over variables $aggrfct(var_i)$ s.t. var_i is bound by a `LET` clause, are replaced by "$aggrfct\{\ var_i\ [var_1, \ldots, var_{i-1}];\ xpath\text{-}expr_i \rightarrow var_i\}$.

Proposition 8.1 (Variable Bindings in XPathLog and XQuery)
The XPathLog rule body constructed above produces the same variable bindings as the XQuery `FLW` statement *except of the variables which are bound to node sets in a* ***LET*** *clause.* □

Proof. By induction over the number of `FOR/LET` clauses: $n = 1$ is obvious, the step from n to $n+1$ uses the variable bindings of previous steps. □

If the `RETURN` clause does not contain variables which are bound to node sets in a `LET` clause, the *xml-pattern* in the `RETURN` clause can be rewritten into a suitable complex XPathLog rule head, inserting the corresponding structures for all variables which are bound to individual nodes. Due to the above proposition, the inserted tree fragments are the same as for XQuery.

Variables bound by `LET` statements to *node sets* which occur in the `RETURN` clause must be handled by successive rules, adding *each* element of the result set separately to the result tree.

Nested Expressions. Similar to XML-QL, nested expressions in the `FLW` clause have to be translated into rules which are executed *before* for creating intermediate result trees. If the nested query is not too complex, it is also often possible to rewrite it and incorporate it into the rule body.

For the `RETURN` clause, the same restrictions as for XML-QL hold: Nested `FLWR` clauses have to be postponed to later updates of the database. The handling of variables which are bound to node sets has also to be postponed to a second, refining step.

Translation from XPathLog to XQuery.

XPathLog reference expressions are atomic components of XPathLog queries.

Proposition 8.2 (Mapping XPathLog Reference Expressions to XQuery queries)

- Every XPathLog reference expression *refExpr* can be translated into an equivalent XQuery `FOR`* `WHERE` expression which binds all variables occurring in *refExpr*.
- Every conjunction of XPathLog reference expressions can be translated into an equivalent XQuery `FOR`* `- WHERE` expression. □

Proof Sketch. For the first step, the reference expression is stepwise resolved into subexpressions similar to the atomize mapping. Note that join variables and filter conditions can optionally be handled in the `WHERE` clause. For the second step, the `FOR` clauses are collected. The resulting `WHERE` clause collects the generated `WHERE` clauses and handles cross-literal join variables (although these sometimes can also be included into the `FOR` clauses). □

Proposition 8.3 (Mapping XPathLog Reference Expressions to XQuery `RETURN` clauses) Every XPathLog reference expression (in the head of an XPathLog rule) which does not use sibling axes can be translated into an equivalent XQuery `RETURN` clause. □

Proof Sketch. Rewrite it into a suitable XML pattern (note that this is not possible for sibling axes). □

Since an individual XPathLog rule cannot create a nested structure with grouping (instead, a sequence of rules has to be used where each rule adds subelements or attributes to the elements generated by previous rules), the translation does not need nested `FLWR` expressions in the `RETURN` clause.

Recursive Queries. XQuery has the same problem as SQL: it cannot express recursive queries, e.g., creating a tree which represents a transitive closure. With XPathLog which comes with iteration as built-in in its fixpoint semantics, such computations are possible.

Thus, although, each *individual* rule can be mapped into an XQuery `FLWR` statement, the *semantics* of XPathLog programs is not covered by XQuery. Note that YAXQL [Moe00] provides a construct for recursive queries.

XQuery does also not (yet) provide an *update* clause in the style of the SQL `UPDATE - SET - WHERE`; a proposal will be published in [TIHW01] (cf. Section 3.12).

8.1.5 XSLT

XSLT is a pattern-based XML transformation language: Similar to XPath rules, each XSLT pattern contributes to the final result. XSLT is less declarative than XML-QL, Quilt, or XPathLog which declaratively specify the outcome of a query or transformation. Instead, XSLT uses explicit apply-pattern, if-then, and for-each commands to control the generation process. Since XSLT employs XPath as addressing mechanism, there is no implicit dereferencing; the id function has to be used.

As already stated in Section 3.4, one of the main drawbacks of XSLT as a database language is that it is not possible to *update* the result tree. When an element of the result tree is constructed, it is not possible to refine it later. Here, different implementations provide the (yet proprietary) nodeset() extension which allows to access the result tree generated so far as an input source. For "real" data integration, it would be necessary to run a sequence of XSLT stylesheets, each one using the result of the previous one and the original sources, which is ineffective since the whole result tree is copied in each step.

As a conclusion wrt. XSLT, XSLT is suitable for handling *documents*, transforming them, merging them, etc., when the result document can be "collected" in one turn. Here, the complexity is low, and the possible cases are easy to grasp. In contrast, in *data integration*, there are more complex cases, and exceptions. Thus, an *incremental* process is more suitable.

Excelon [eXc] extends XSLT with an update language *XUL* (XML Update Language, cf. Section 3.13) which allows to specify updates on elements which are selected by XPath expressions.

8.2 Order

As stated in Section 4, the order of elements in the result tree is relevant from the document point of view, whereas in the database area, it can often be dropped for the benefit of optimizations.

The XML querying and transformation languages which have been described above use an order-preserving semantics. For the "pioneer" languages XQL and XML-QL which have been presented before the XPath proposal, their semantics has been defined in this way "from scratch". The more recent languages (e.g., XSLT and Quilt/XQuery) are syntactically based on XPath which does *not* impose any constraints on the order of element in its result sets (although most of the implementations are order-preserving), but semantically they presume *result lists* which contain the elements in document order – thus, implementations rely on an implementation-dependent, optional "feature" (or they have to reorder the result set according to document order by themselves).

- XQL queries are order-preserving if no explicit sequencing is specified. It is not based on XPath (in contrast, it is one of the "predecessors" of XPath).
- XSLT uses XPath for selecting elements for applying patterns by its <xsl:apply-templates select="*xpath-expr*"> elements. Here, it is implicitly assumed that *xpath-expr* returns the elements in document order.
- XML-QL is also not based on XPath, but on evaluating XML patterns. There exists an order-preserving semantics, and a non-order-preserving semantics.
- The XML Query Algebra [XMQ01a, Section 2.8] explicitly distinguishes an ordered and an unordered model.
- Quilt/XQuery use XPath in the `FOR - LET` clauses. Here, the nesting of the clauses determines the order of elements in the result. Again, it is assumed that the XPath expressions return the elements in document order.
- The XML update operators proposed in [TIHW01] also distinguish between ordered and unordered semantics.

When regarding a graph database, even the notion of order becomes ambiguous:

1. Global, document order: all nodes in a tree are ordered globally. This is the case for the DOM and XML Query Data Model. The extension to multiple trees in a database is not clear (seems to be arbitrary, depending on the order of parsing documents).
2. With the multi-tree overlapping model and restructuring of documents by linking (cf. Section 11), the global ordering is sometimes too strong. Thus, XPathLog enumerates the children of a node locally, not imposing a global order (see Section 5.3). As a consequence, it is e.g. allowed that e_1 is the first child of a node x, and e_2 is the second child, whereas for another node y, e_2 is the first child and e_1 is the second child. This may be useful when the contents of a source has to be reordered.
3. When navigating along reference attributes, another "navigation order" arises, enumerating elements in the order they are "found" by the evaluation: An `IDREFS` attribute is itself an ordered list of references, and it can be useful to return the referenced nodes in the order of the referencing nodes (i.e., where for //organisation/@seat→city, the cities are enumerated as a sequence consisting of the seat of the first organization, then the seat of the second organization, etc.).

The current proposals all use (1). XPath does not specify any order of the result (always returning result sets). Thus, applications based on XPath have to use the global document order. Especially, XSL(T), XML-QL, and Quilt/XQuery, return the result of applying the id() function according to the order of the *referenced* nodes in the document.

In contrast, the semantics definition given for XPathLog in Sections 5 and 6 use the above "navigation order" which preserves the local order of children and of items in reference attributes. If additionally a global document order is given, the results can be reorderd accordingly.

8.3 XPathLog vs. XML-QL Data Model

The XPathLog data model by X-structures has some similarities with the XML-QL data model (cf. Section 3.5): Both (the XML-QL data model has been influenced by the STRUDEL/STRUQL [FFK+98] project) emerged from general semi-structured models. Both are graph-based (instead of strictly tree-based) and edge-labeled (with a single type of properties/relationships), but originally unordered. For both models, references based on *object identity* are a native built-in concept of the data model. On the other hand, the migration to the XML world required to extend the models to distinguish attributes and elements, and to augment the subelement relationship with ordering. For XML-QL, the ordered model extends the unordered one by associating an order of the *complete* set of nodes (i.e., the order of the outgoing edges of (i.e., children) of each node is induced by this global order). In contrast, in the XPathLog data model, the order of the subelements is associated with the parent node.

In contrast to tree models, the graph models allow – at least theoretically – for having more than one parent of an element. The *pure* XML tree model does not use this opportunity, so the data models are somehow "too rich" – but they *cover* the XML model. With XPathLog, the possibility of having several parents is exploited in Sections 11 and 12 for XML data integration: it allows for generating an *XML database* which contains multiple overlapping trees and *tree views* from several sources and then to distinguish *result views* as XML trees.

8.4 XPathLog vs. XML Query Data Model

The XML Query Data Model is a special case of a W3C Requirement: a data model is specified which is intended to serve as a formal background for XML querying languages. Some constraints (see Section 3.9.2) conflict with the requirements for data restructuring and integration.

$\mathcal{X}$-structures have been defined in Section 5.3 as the formal data model underlying XPath-Logic, based on the notion of a *navigation graph* (cf. Definition 5.4). For an XML document, the induced navigation graph contains the XML tree and cross edges which represent reference attributes.

In contrast to the XML Query Data Model,

- XPathLog does not use *text nodes*, *attribute nodes*, and *reference nodes* (note that the actual semantics of reference nodes is also not defined in the XML Query Data Model, except that they may be used for handling reference attributes – the XPathLog data model does not need such auxiliary constructs). Instead, there are *literal values* which are used to represent text children and literal-valued attributes. Attributes are directly represented by (sets of) literals or references to element nodes.

- the navigation graph is *edge-labeled*, i.e., the names of the subelements and attributes are annotated to the *edges*, not to the nodes. This allows for defining overlapping tree views where the same elements may occur under different names

- arbitrary "subelement" edges from element nodes to other element nodes can be added to the graph (thus, even the subelement relationship does not necessarily induce a tree).

Example 8.2 (Overlapping Tree Views)
The database contains the MONDIAL *XML tree which includes a subelement edge labeled with* city *from the element node* germany *to the element node representing* berlin*. Additionally, the user may define a "metropole view" on the database by*

```
result[metropole→C] :- cia/city→C[population > 1000000].
```

which adds a subelement edge labeled with metropole *from* result *to* berlin *(cf. Figure 10.1).*

Thus, the navigation graph is not intended to represent "the XML document", but a database where XML documents may be defined as "tree views" – some of the views are the original XML trees (see Section 11).

8.5 Data Manipulation in the DOM/XML Query Data Model

As already stated in Section 3.9.2 and 8.4, the DOM and XML Query Data Model severely restrict the design of XML data *manipulation* languages. The XPathLog data model is a compatible *extension* to the XML/DOM data model, i.e., the "restriction" of XPathLog to the features supported by the node-labeled tree models is a well-defined querying and restructuring language for XML *trees*:

- XPathLog rule bodies are a complete XML querying language which allows for querying all properties of XML documents in the DOM model.
- Since the XML/DOM Data model requires elements to have at most one parent node, it is not possible to restructure a given XML tree in-place. Instead, any notion of restructuring involves the creation of a result tree by *deep-copying* elements (as done implicitly by XSLT, XML-QL, and Quilt/XQuery). Note that obviously, the same IDs occur in the original tree and in the result tree.

  For these and similar problems, the DOM model provides *reference nodes*. The Xerces DOM implementation (which has been used in [Beh01] for the DOM-based implementation variant of LoPiX described in Section 15.2) automatically maintains copied trees to point their references also in the copied tree – which is a solution as long as the reference target also belongs to the fragment which is deep-copied. Copying cross-references in Xerces yields unintended results.
- Thus, for a comparison, the semantics of XPathLog heads must be (temporarily) restricted and redefined as follows:
  - rule heads may only update the result tree, i.e., all elements occurring as start nodes of expressions in rule heads are required to belong to the result tree,
  - nodes which are bound to variables are deep-copied when instantiating the rule head,
  - during the computation, the result tree may contain dangling reference attributes. As long as there is no target of the reference attribute in the result tree, it has to be evaluated wrt. the corresponding element in the original tree (which is guaranteed to exist since the original tree is not changed).

    We allow for queries accessing the temporary result tree, corresponding to use variables bound by previous `FOR/LET` clauses in Quilt/XQuery.
  - after the computation, all reference attributes in the result tree have to be checked (i.e., it has to be checked if `result` is a root according to Definition 5.6).

    Note that this is also the case for XML-QL and Quilt/XQuery – their results can also contain dangling references.

With the above definitions, XPathLog can be implemented based on a DOM API implementation (cf. Section 15.2). Nevertheless, maintaining reference attributes during complex restructuring and integration tasks yields difficult problems:

Example 8.3 (Updates with Restrictions)
Assume two trees A and R(esult), where subtrees A_1 and A_2 which reference each other will be copied during the process to R.

First, A_1 is copied from A to A_1' in R, duplicating it. It is reasonable to adapt all references inside the copied fragment A_1 to A_1'. Since A_2 is not yet copied (even not known to be copied

later), references from A'_1 to A_2 cannot be adapted; they still point to A_2 in A. Wrt. R they are dangling.
Then, some other operations are done.
Later, A_2 is copied to A'_2 somewhere in R, its internal references are adjusted. Now, *references from A'_1 in R to nodes in A_2 could be adjusted to A'_2:*

> *"adjust all references in the target tree which point into the subtree which is copied."*

This is not correct in the general case when a tree R_1 is copied inside R to R_2 – here only the internal references should be adjusted. Refine the above with

> *"... if the subtree to be copied is not part of the target tree."*

Now, A'_2 in R contains references to A_1 in A – wrt. R, they are dangling now.

How is it possible to adjust the references from A_2 to A_1 into references from A'_2 to A'_1? Note that A_1, A_2, and A'_1 may be changed in the meantime.

Thus, by using an *edge-labeled* graph instead of the *node-labeled* graph in the XML data model, and refraining from the "unique-parent" constraint of the DOM and XML Query Data Model, XPathLog gains its expressiveness as an XML data manipulation and integration language. The proposal for updating XML in [TIHW01] has to solve the copying problem described above.

8.6 An XML Syntax for XPathLog

The XML Query Requirements [XMQ01c] demand for an XML-syntax language binding for an XML querying/manipulation language. Similar to XSLT and YAXQL, elements with the semantics of the required constructs, the notions of *rule*, *stratum*, and *program* can be defined. The xpathlog: namespace contains the following elements:

Program: a program is a sequence of strata with optional querey in-between,

Stratum: a stratum is a sequence of rules and facts in arbitrary order. For optimization, it may be specified if it is sufficient to evaluate all rules once (one TX_P round), or if the full TX_P^∞ computation is needed.

Rule: a rule consists of a head and a body.

Head: a head is a list of atoms.

Body: a body is a list of literals.

Atom: an atom has a body which is an XPathLog expression (`CDATA`).

Literal: a literal may be negated and has a body which is an XPathLog expression (`CDATA`).

Fact: a fact is an atom. It is sufficient to insert it once.

Query: a query consists of a set of literals.

The elements of the language are specified by the following DTD:

```
<!ELEMENT program ((stratum|query?)+)>
<!ELEMENT stratum ((fact|rule)*)>
    <!ATTLIST stratum once (YES|NO) DEFAULT "NO">
<!ELEMENT rule (head,body)>
    <!ATTLIST rule once (YES|NO) DEFAULT "NO">
```

```
<!ELEMENT head (atom+)>
<!ELEMENT body (literal+)>
<!ELEMENT atom EMPTY>
      <!ATTLIST atom body CDATA #REQUIRED
<!ELEMENT literal EMPTY>
      <!ATTLIST literal negation (YES|NO) DEFAULT "NO"
                           body CDATA #REQUIRED>
<!ELEMENT fact (atom) EMPTY>
      <!ATTLIST fact once (YES|NO) "YES">
<!ELEMENT query (literal+)>
```

The elements may e.g. be augmented with information about program details which serve for optimization or semantical checks. For example, atoms, literals, heads and bodies can have additional attributes *vars* and *safevars*. A rule is safe, if all variables in the head occur positively in the body. Additionally, such information can be used for syntactical goal reordering. Perhaps the XML syntax is suitable to make people like rule-based programming who do not like Prolog notation. A more general approach for representing inference rules in XML is described in [RML01].

9 CLASS HIERARCHY AND INHERITANCE

XPathLog is extensible *in two orthogonal directions:*

- *Extending the approach to multiple trees and tree* views *of the XML database, combined with a projection mechanism.*
- *The data model can be extended by additional conceptual notions, e.g., a class hierarchy, or signature information which can also be integrated into the language.*

The following sections describe extensions to the basic XPathLog language. Thus, the presentation is given mainly on the level of DOM Herbrand structures instead of X-structures.
An extension to a class hierarchy and inheritance is presented in Section 9. Then, a lightweight signature concept using signature specifications *is defined in Section 10. Both extensions are adaptations from the experiences in data modeling with F-Logic/*FLORID *[LHL+98, MHLL99].*
The signature is especially important for defining individual tree views as projections from the database. The extension to multiple XML trees and data integration is described in Section 11, followed by the case study in Section 12. The LoPiX *system which implements XPathLog with these extensions is presented in Section 15.*

Types or classes (in general organized with a class hierarchy and equipped with some inheritance concept) provide an important means for *data modeling*. Additionally, for data integration, reasoning on the meta level, i.e., signatures (possibly augmented by ontologies which provide meta information about signatures) is a crucial requirement.

The pure XML data model does not directly include a notion of classes and class membership. For XML, signature and structure information is usually given either by DTDs or by XML Schema instances. For the XPathLog data model, we renounce from defining a special extension for describing structure, but restrict ourselves to a "lightweight" representation of signature, i.e., on the relationships between types and properties (cf. Section 10).

In XPathLog, the signature provides a metadata description which can be used in queries, updated by rule heads, and especially serves for defining *projections* of the database to one or more result trees which thus can be regarded as *views* on the whole database (cf. Section 11).

9.1 Classes

The element types induce a notion of classes:

- Elements are complex types, thus it makes sense to regard them as objects:
  - every element type t defines a class which is a subclass of the most general object class *object*,
  - for every instance e of element type t, e is a t.
- text contents and attribute values are simple types (recall that XPath-Logic silently splits `NMTOKENS` into their individual values): Every such value is a member of the class *literal* which is further partitioned into *numeric* and *string*.

Here (as long as no signatures are used, cf. Section 10), the subclass relationship just induces transitivity of subclass relation and class membership.

From the formal point of view, the X-structures are extended:

Definition 9.1 (X-Structure with Classes)
An *X-structure with classes* is an X-structure (cf. Definition 5.3) extended as follows:

- the signature contains a set Σ_{Cl} of class names (often identical with the element type names),
- $\mathcal{N}$ additionally contains two (disjoint) sets $\mathcal{N}_{\mathcal{CE}}$ and $\mathcal{N}_{\mathcal{CL}}$ of *element class names* and *literal class names*,
- X-structures (cf. Definition Def-canonical-x-structure) interprete class names by

$$\mathcal{I}_{\mathcal{C}} : \Sigma_{Cl} \to \mathcal{N}_{\mathcal{CE}} \cup \mathcal{N}_{\mathcal{CL}}$$

- the class membership is represented by $\mathcal{C}$ as two (partial) mappings

$$\mathcal{C}_{\mathcal{V}} \; : \; \mathcal{V} \to 2^{\mathcal{N}_{\mathcal{CE}}} \; ,$$

  mapping each node/vertex n to a set C of object classes denoting that n belongs to all classes in C, and

$$\mathcal{C}_{\mathcal{L}} \; : \; \mathcal{L} \to 2^{\mathcal{N}_{\mathcal{CL}}} \; ,$$

  mapping each literal ℓ to a set C of literal classes denoting that ℓ belongs to all classes in C, and

- the subclass relationships are represented by $\mathcal{SC}$, also as two (partial) mappings

$$\mathcal{SC}_{\mathcal{E}} \; : \; \mathcal{N}_{\mathcal{CE}} \to 2^{\mathcal{N}_{\mathcal{CE}}} \; ,$$

  mapping each object class c to a set C of object classes, denoting that c is a subclass of all classes in C, and

$$\mathcal{SC}_{\mathcal{L}} \; : \; \mathcal{N}_{\mathcal{CL}} \to 2^{\mathcal{N}_{\mathcal{CL}}} \; ,$$

  mapping each literal class c to a set C of literal classes, denoting that c is a subclass of all classes in C.

- $\mathcal{C}$ and $\mathcal{SC}$ satisfy a closure requirement (transitivity of the class hierarchy):
  - if $d \in \mathcal{SC}(c)$ and $c \in \mathcal{SC}(b)$, then $d \in \mathcal{SC}(b)$, and
  - if $d \in \mathcal{SC}(c)$ and $c \in \mathcal{C}(b)$, then $d \in \mathcal{C}(b)$. □

The canonical X-structure to an XML document is also extended with the class information. Here, it depends whether the class information is derived from the XML instance or the DTD, or from an XML Schema description.

Definition 9.2 (Canonical X-Structure with Classes for XML and DTD)
Given an XML instance $\mathcal{D}$, the canonical X-structure $\mathcal{X}_{\mathcal{D}}$ (see Definition 5.5) is extended with an interpretation of $\mathcal{C}$ and $\mathcal{SC}$. $\mathcal{N}_{\mathcal{E}} = \mathcal{N}_{\mathcal{CE}}$ is the set of element names occurring in the XML instance or in its DTD.

$$\begin{array}{rcl} \mathcal{C}(v) & = & \begin{cases} \{name\} & \text{if } v \in \mathcal{V} \text{ and } \mathsf{name}(\Phi^{-1}(v)) = name \\ \{\mathsf{literal}\} & \text{where } v \in \mathcal{L} \text{ represents a text node or an } \texttt{NMTOKEN} \text{ or} \\ & \text{a } \texttt{CDATA} \text{ attribute value} \end{cases} \\ \mathcal{SC}(x) & = & \{\mathsf{object}\} \text{ if } x \in \mathcal{N}_{\mathcal{CE}} \text{ is an element class.} \end{array}$$

□

The definition exploits the fact that element and attribute names are full citizens of the model which may also occur at host and result positions of atoms and in predicates.

If an XML Schema instance is given, the class hierarchy does not use the element names, but the element types (*simpleTypes* and *complexTypes*; see Section 10.2.2); then, $\mathcal{N}_\mathcal{E}$ and $\mathcal{N}_{\mathcal{CE}}$ are in general not the same.

Example 9.1 (X-Structure with Classes)
The X-structure (see Example 5.5) of the XML instance given in Example 5.1 contains the following class memberships:

$$\begin{array}{ll}
\mathcal{C}(\textit{mondial}) = \{\textit{mondial}\}, & \mathcal{C}(\textit{un}) = \{\textit{organziation}\}, \\
\mathcal{C}(\textit{ch}) = \{\textit{country}\}, & \mathcal{C}(\textit{un-name}) = \{\textit{name}\}, \\
\mathcal{C}(\textit{ch-name}) = \{\textit{name}\}, & \mathcal{C}(\textit{un-abbrev}) = \{\textit{abbrev}\}, \\
\mathcal{C}(\textit{bern}) = \{\textit{city}\}, & \mathcal{C}(\textit{un-mem-mem}) = \{\textit{member}\}, \\
\mathcal{C}(\textit{bern-pop}) = \{\textit{population}\}, & \mathcal{C}(\textit{un-mem-obs}) = \{\textit{member}\},
\end{array}$$

$$\mathcal{SC}(\mathsf{class}) = \{\textit{object}\} \text{ for all the above classes.}$$

and $\mathcal{C}(\ell) = \{\textit{literal}\}$ *for all text contents and attribute values* $\ell \in \mathcal{L}$.

Class Hierarchy in XPath-Logic. XPath-Logic seamlessly integrates class membership and subclass information via *class atoms* which are special (infix) predicates satisfying the above closure property:

- *XPath-Logic subclass atoms* are of the form

  $class_1$ subcl $class_2$

  which denotes that the class $class_1$ is a subclass of $class_2$:

- *XPath-Logic class membership atoms* are of the form

  $node$ isa $object\_class$ resp. $value$ isa $literal\_class$

  which denotes that the element *node* is a member of the object class *object_class* (which in turn is a subclass of object), or that *value* is a member of the class *literal_class*.

Concerning the DOM Herbrand structure, we write $A \in \mathcal{HD}$ for isa/class atoms instead of $A \in \mathsf{preds}(\mathcal{HD})$ since they represent built-in notions.
The formal definition of the semantics of formulas is extended appropriately:

Definition 9.3 (Semantics of XPath-Logic Class Atoms)
The semantics of XPath-Logic class atoms is defined as follows in the style of special predicates:

- interpretation of class names: for $cl \in \Sigma_{Cl}$, define $\mathcal{S}_\mathcal{X}(cl,\beta) := \mathcal{I}_\mathcal{C}(cl)$ for all variable assignments β.
- truth values (cf. Definitions 5.9 and 5.10):

$$\begin{array}{lll}
(\mathcal{X},\beta) \models expr_1 \ \mathsf{subcl}\ expr_2 & :\Leftrightarrow & \mathcal{S}_\mathcal{X}(expr,\beta) \in \mathcal{SC}(\mathcal{S}_\mathcal{X}(expr,\beta)) \\
(\mathcal{X},\beta) \models expr_1 \ \mathsf{isa}\ expr_2 & :\Leftrightarrow & \mathcal{S}_\mathcal{X}(expr,\beta) \in \mathcal{C}(\mathcal{S}_\mathcal{X}(expr,\beta))
\end{array}$$

- answer variable bindings (cf. Definition 6.8; similar to predicates)

$$\mathcal{QB}(expr_1 \ \mathsf{subcl}\ expr_2, Bdgs) := \bigcup_{\substack{(x_1,\xi_1)\in\mathcal{SB}^{any}_{\mathcal{HD}}(expr_1,Bdgs),\ldots,\\ (x_n,\xi_1)\in\mathcal{SB}^{any}_{\mathcal{HD}}(expr_2,Bdgs),\\ (x_1 \mathsf{subcl} x_2)\in\mathcal{HD}}} \xi_1 \bowtie \xi_2$$

  analogous for isa . □

The transitivity of $\mathcal{C}$ and $\mathcal{SC}$ is maintained by the $\mathcal{TX}_P$-Operator:

Definition 9.4 (Closure Axioms: Extended $\mathcal{TX}_P$-Operator)
For a DOM Herbrand Structure $\mathcal{HD}$, the *closure* operator, $\mathcal{C}\ell(\mathcal{HD})$ is defined as

- if u_1 subcl $u_2 \in \mathcal{HD}$ and u_2 subcl $u_3 \in \mathcal{HD}$ then u_1 subcl $u_3 \in \mathcal{C}\ell(\mathcal{HD})$ (subclass transitivity),
- if u_1 subcl $u_2 \in \mathcal{HD}$ and u_2 subcl $u_1 \in \mathcal{HD}$ then $u_1 = u_2 \in \mathcal{C}\ell(\mathcal{HD})$ (subclass acyclicity),
- if u_1 isa $u_2 \in \mathcal{HD}$ and u_2 subcl $u_3 \in \mathcal{HD}$ then u_1 isa $u_3 \in \mathcal{C}\ell(\mathcal{HD})$ (instance-subclass dependency),

The $\mathcal{TX}_P$-Operator (cf. Definition 7.5) is extended to a $\mathcal{TX}_{P,\mathcal{C}\ell}$-Operator:

$$\mathcal{TX}_{P,\mathcal{C}\ell}(\mathcal{HD}, B) := \mathcal{C}\ell(\mathcal{TX}_P(\mathcal{HD}, B))$$

where

$$\mathcal{C}\ell(\mathcal{HD}, B) := (\mathcal{C}\ell(\mathcal{HD}), B)$$

completes the DOM Herbrand structure component and leaves the dictionary of bindings unchanged. □

Note that in an implementation, the $\mathcal{C}\ell$ operator and the class memberships of literals are not necessarily implemented by materializing the derived atoms. The LoPiX implementation (cf. Section 15) does not materialize the closure, but includes it into the evaluation of queries and rule bodies.

When using a DTD, classes provide an alternative specification of all elements of a given element type. A class c at least contains all nodes n such that $\mathsf{name}(n) = c$. For the canonical DOM Herbrand structure $\mathcal{HD}$ to an XML document, the answer set (cf. Definition 6.11) to an isa query yields all these elements:

$$\mathsf{answers}_{\mathcal{HD}}(//name{\rightarrow}\mathsf{C}) = \mathsf{answers}_{\mathcal{HD}}(\mathsf{C}\ \mathsf{isa}\ name)$$

Additionally, the class hierarchy can be changed by XPathLog rules. Note that with overlapping trees, a node may belong to several classes (wrt. different *tree views*).

9.2 Class Hierarchy and Inheritance

In the basic XML model, the notion of fixed and default values of attributes defined in the DTD provides a very restricted notion of *monotonic and nonmonotonic inheritance*. In the object-oriented data model, inheritance is combined with a *class hierarchy*. After extending XML with a class hierarchy in the previous section, the inheritance model is described now.

Definition 9.5 (X-Structures with Inheritance)
For X-structures, the mapping $\mathcal{A}$ (attributes) is extended with two more (partial) mappings

$$\mathcal{A}^* \;:\; \mathcal{N}_{\mathcal{CE}} \times \mathcal{N}_{\mathcal{A}} \rightarrow 2^{\mathcal{V}} \cup 2^{\mathcal{L}} \quad \text{and} \quad \mathcal{A}^{**} \;:\; \mathcal{N}_{\mathcal{CE}} \times \mathcal{N}_{\mathcal{A}} \rightarrow 2^{\mathcal{V}} \cup 2^{\mathcal{L}}$$

which specify the default ($\mathcal{A}^*$) and fixed ($\mathcal{A}^{**}$) values. $v \in \mathcal{A}^*(class, attrname)$ denotes that v is a default value of attribute *attrname* of elements of class *class*[1].

The monotonic inheritance semantics of `FIXED` attributes adds another closure requirement (whereas for non-monotonic inheritance, the semantics of Default Logics applies, which is added to the evaluation, see Section 9.2.1):

[1]for a multivalued default, $\mathcal{A}^*$ may contain several elements.

- if $v \in \mathcal{A}^{**}(class, attr)$ and $class \in \mathcal{SC}(sub)$, then $v \in \mathcal{A}^{**}(sub, attr)$ (propagation to subclasses), and
- if $v \in \mathcal{A}^{**}(class, attr)$ and $class \in \mathcal{C}(x)$, then $v \in \mathcal{A}(x, attr)$ (propagation to instances). □

In contrast to $\mathcal{A}$, $\mathcal{E}$, and $\mathcal{C}$, the extension of these mappings is not contributed by the XML instance, but by the DTD (or by the XML Schema definition, see Section 10.2.2):

Definition 9.6 (Canonical X-Structure with Inheritance (DTD))
Given an XML instance $\mathcal{D}$ using a DTD dtd, the canonical X-structure $\mathcal{X}_\mathcal{D}$ (cf. Definitions 5.5 and 9.2) is extended with an interpretation of $\mathcal{C}$, $\mathcal{A}^*$, and $\mathcal{A}^{**}$:

- For an element declaration

  <!ELEMENT *element contentsmodel*> ,

  object $\in \mathcal{SC}$(*element*), and

- for attribute declarations,

$$\mathcal{A}^*(name, attr) = \begin{cases} \{v \mid \mathcal{A}(v, attr') = ref_i \text{ where } attr' \text{ is the ID attribute of } v \\ \quad \text{if } dtd \text{ specifies} \\ \quad \text{<!ATTLIST } name\ attr \text{ IDREF(S) ``}ref_1 \ldots ref_n\text{'' > } \} \\ \{\ell \mid \ell \text{ is a value item in } value \text{ if } dtd \text{ specifies} \\ \quad \text{<!ATTLIST } name\ attr\ attr\_type \text{ ``}value\text{'' >} \\ \quad \text{as a non-reference attribute. } \} \end{cases}$$

  Analogously for $\mathcal{A}^{**}$ and `FIXED`. □

Example 9.2 (DTD, Default and Fixed values cont'd)
Consider again the DTD fragment given in Example 2.4.

<!ELEMENT *name* EMPTY>
<!ATTLIST *name* $attr_1$ CDATA "*value*"
$attr_2$ IDREFS #FIXED "id_1 id_2">

As already mentioned, a valid XML document which contains some node n of type name must provide some nodes n_1 and n_2 with `ID` *attributes a_1 and a_2 and such that $n_i[@a_i = id_i]$ as reference targets. Then,*

$$\mathcal{A}^*(name, attr_1) = \{value\} \quad and \quad \mathcal{A}^{**}(name, attr_2) = \{n_1, n_2\} .$$

In the Herbrand structure and in XPathLog, $\mathcal{A}^*$ and $\mathcal{A}^{**}$ are represented by atoms similar to the atoms for specifying attribute values and subelement relationships:

elementtype[@*attribute* •→*result*] and *elementtype*[@*attribute* ••→*result*]

The semantics $\models$ and $\mathcal{Q}B$ of these atoms are defined straightforwardly based on evaluating terms and checking $\mathcal{A}^*$ and $\mathcal{A}^{**}$, or $\mathcal{H}D$, respectively.

9.2.1 Evaluation with Inheritance

The monotonic inheritance semantics of `FIXED` attributes is directly added to the closure operator, whereas non-monotonic inheritance of defaults is added to the evaluation. In [MK98, MK01], it has been shown that this semantics coincides with the standard semantics of Default Logic (which then also applies to the X-structure level).

Definition 9.7 (Closure Axioms: Extended $\mathcal{TX}_P$-Operator)
Extending Definition 9.4, for a DOM Herbrand Structure $\mathcal{HD}$, the *closure* operator, $\mathcal{C}\ell(\mathcal{HD})$ is redefined as

- if u_1 subcl $u_2 \in \mathcal{HD}$ and u_2 subcl $u_3 \in \mathcal{HD}$, then add u_1 subcl u_3 (subclass transitivity),
- if u_1 subcl $u_2 \in \mathcal{HD}$ and u_2 subcl $u_1 \in \mathcal{HD}$, then add $u_1 = u_2$ (subclass acyclicity),
- if u_1 isa $u_2 \in \mathcal{HD}$ and u_2 subcl $u_3 \in \mathcal{HD}$ then add u_1 isa u_3 (instance-subclass dependency),

▷ if $c[@a{\bullet\!\bullet\!\!\rightarrow}v] \in \mathcal{HD}$ and c' subcl $c \in \mathcal{HD}$ then add $c'[@a{\bullet\!\bullet\!\!\rightarrow}v]$ (monotonic inheritance to subclasses),

▷ if $c[@a{\bullet\!\bullet\!\!\rightarrow}v] \in \mathcal{HD}$ and *node* isa $c \in \mathcal{HD}$ then add (v, a) to $\mathcal{A}_{\mathcal{HD}}(\mathsf{attribute}, node)$ (monotonic inheritance to instances),

$**$ if $c[@a{\bullet\!\bullet\!\!\rightarrow}v] \in \mathcal{HD}$ and c' subcl $c \in \mathcal{HD}$, then there is no w s.t. $c'[@a{\bullet\!\bullet\!\!\rightarrow}w] \in \mathcal{HD}$ and not $c[@a{\bullet\!\bullet\!\!\rightarrow}w] \in \mathcal{HD}$,

$**$ if $c[@a{\bullet\!\bullet\!\!\rightarrow}v] \in \mathcal{HD}$ and *node* isa $c \in \mathcal{HD}$, then there is no w s.t. $(w, a) \in \mathcal{A}_{\mathcal{HD}}(\mathsf{attribute}, node)$ and not $c[@a{\bullet\!\bullet\!\!\rightarrow}w] \in \mathcal{HD}$,

- if one of the $(**)$ conditions is violated, then $\mathcal{C}\ell(\mathcal{HD}) = \perp$, i.e., $\mathcal{HD}$ is inconsistent.

Again, the $\mathcal{TX}_{P,\mathcal{C}\ell}$-Operator defines the semantics of a program. □

For the semantics of *non-monotonic inheritance*, the semi-declarative semantics which has been defined for F-Logic in [KLW95] is adopted and implemented in FLORID [FLO98, FHK+97]; it has been carried over to LoPiX. In [MK98, MK01], it has been shown that this semantics coincides with the standard semantics of Default Logic [Poo94] and Inheritance Networks [Hor94]. Non-monotonic inheritance is implemented via a trigger mechanism in a *deduction precedes inheritance* manner: The evaluation of a program is defined by alternatingly computing a classical deductive fixpoint and carrying out a specified amount of inheritance. The strategy is formally characterized as follows, based on *inheritance triggers* [KLW95], for proofs see [MK01]:

Definition 9.8 (Inheritance Triggers)
Let $\mathcal{HD}$ be a DOM Herbrand structure.

- An *inheritance trigger* in $\mathcal{HD}$ is a pair $(n\sharp c, @a{\bullet\!\!\rightarrow}v)$ such that $(n\sharp c) \in \mathcal{HD}$ and $c[@a{\bullet\!\!\rightarrow}v] \in \mathcal{HD}$, and there is no $n \neq c' \neq c$ such that $\{n\sharp c', c' \text{ subcl } c\} \subseteq \mathcal{HD}$ (where $\sharp$ stands for isa or subcl).
- An inheritance trigger $(n \text{ isa } c, @a{\bullet\!\!\rightarrow}v)$ or $(c' \text{ subcl } c, @a{\bullet\!\!\rightarrow}v)$ is *active* in $\mathcal{HD}$ if there is no v' such that $(v', a) \in A_{\mathcal{HD}}(\mathsf{attribute}, n)$ or $c'[@a{\bullet\!\!\rightarrow}v'] \in \mathcal{HD}$, respectively.
- $\mathbf{T}(\mathcal{HD})$ denotes the set of active inheritance triggers in $\mathcal{HD}$.
- An inheritance trigger $(n \text{ isa } c, @a{\bullet\!\!\rightarrow}v)$ or $(c' \text{ subcl } c, @a{\bullet\!\!\rightarrow}v)$ is *blocked* in $\mathcal{HD}$ if $(v', a) \in A_{\mathcal{HD}}(\mathsf{attribute}, n)$ or $c'[@a{\bullet\!\!\rightarrow}v'] \in \mathcal{HD}$, respectively, for some $v' \neq v$.

Note that this definition depends only on $\mathcal{HD}$, not on a program. □

An attribute value is inherited from a class to a node or a subclass only if no other value for this attribute can be derived for the node or the subclass, respectively. Hence, inheritance is done *after* classical deduction, leading to an alternating sequence of (deductive) fixpoint computations and inheritance steps.

Definition 9.9 (Firing a Trigger)
For a DOM Herbrand structure $\mathcal{HD}$ and an active trigger $t = (n \text{ isa } c, @a{\bullet\!\!\to} v)$ or $t = (c' \text{ subcl } c, @a{\bullet\!\!\to} v)$, the DOM Herbrand structure after firing t, $t(\mathcal{HD})$, is obtained by adding (v, a) to $\mathcal{A}_{\mathcal{HD}}(\mathsf{attribute}, n)$ or adding $\{c'[@a{\bullet\!\!\to} v]\}$ to $\mathcal{HD}$, respectively.
In accordance to [KLW95], for a DOM Herbrand structure $\mathcal{HD}$ and an active trigger t, $\mathcal{I}^t_P(\mathcal{HD}) := \mathcal{TX}^{\omega}_{P,\mathcal{Cl}}(t(\mathcal{HD}))$ denotes the *one step inheritance transformation*. □

Proposition 9.1 (Correctness of One-Step-Inheritance)
Let P be a program and $\mathcal{HD}$ a DOM Herbrand structure which is a model of P (i.e., $\mathcal{HD} \models h \leftarrow b$ for every rule in P). For every $t \in \mathbf{T}(\mathcal{HD})$, if $\mathcal{I}^t_P(\mathcal{HD}) = T^{\omega}_P(t(\mathcal{HD}))$ is consistent, then it is also a model of P. □

Note that the notion of a model of an XPathLog program does not require closure wrt. inheritance. Analogous to [KLW95] for F-Logic, *inheritance-canonic* models of XPathLog programs are defined:

Definition 9.10 (Inheritance-Canonic Model)
For an XPathLog program P, a sequence $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_n$ of DOM Herbrand structures is an $\mathcal{I}_P$*-sequence* if $\mathcal{M}_0 = T^{\omega}_P(\emptyset)$ and for all i, there is a $t_i \in \mathbf{T}(\mathcal{M}_i)$ such that $\mathcal{M}_{i+1} = \mathcal{I}^{t_i}_P(\mathcal{M}_i)$.
An DOM Herbrand structure $\mathcal{M}$ is an *inheritance-canonic* model of P if there is an $\mathcal{I}_P$-sequence $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M} \neq \perp$ such that $\mathcal{M}$ has no active triggers. □

Obviously, an XPathLog program P can have several inheritance-canonic models.

Corollary 9.2 (Correctness of Inheritance-Canonic Models)
Let P be an XPathLog program. Then, for every $\mathcal{I}_P$-sequence $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_n$, every $\mathcal{M}_i$ is a model of P. □

Note that further problems with inconsistent results due to *scalarity requirements* in F-Logic which have been described in [MK01] have not to be considered for XPathLog.
An example will be given below, after defining a syntax how to express subclass relationships in a DTD.

Remark 9.1
From the database point of view (i.e., without considering the order of subelements and using a multi-parent model), it is also possible to extend inheritance to the subelement relationship. □

9.2.2 Proposal: Class Hierarchy in XML/DTD

```
<!SUBCLASS elementtype elementtype (contentsdecl)>
```

where contentsdecl is optional. If no contents declaration is given, the subclass inherits the contents model from the superclass; otherwise the contents model of the subclass is required to refine the contents model of the superclass (see Sections 10.3 and 10.4). A subclass may also have an <!ATTLIST ...> declaration which then extends and refines the attribute definitions of the superclass.
The following example illustrates the subclass hierarchy, nonmonotonic inheritance, and the trigger mechanism:

Example 9.3 (Default Values and Inheritance)
Consider the following DTD "zoo.dtd"

```
<!-- XML DTD "zoo.dtd" -->
<!ELEMENT zoo (animal*)>
<!ELEMENT animal EMPTY>
```

```
<!ATTLIST animal name CDATA #IMPLIED
                 fly CDATA "no">

<!ELEMENT bird EMPTY>
<!ATTLIST bird fly CDATA "yes"
               laying_eggs CDATA "yes">

<!ELEMENT penguin EMPTY>
<!ATTLIST penguin fly CDATA "no">

<!SUBCLASS bird animal>
<!SUBCLASS penguin bird>
```

with a small instance:

```
<!DOCTYPE mondial SYSTEM "zoo.dtd">
<zoo>
   <penguin name="tweety"/>
   <bird name="lora"/>
</zoo>
```

After parsing the XML instance, and evaluating it (i.e., since the program is empty, computing the closure of the class hierarchy and handling inheritance) it can be queried:

```
?- zoo/M[name→N and fly→F].
M/penguin   N/"tweety"   F/"no"
M/bird   N/"lora"   V/"yes"
```

The evaluation process proceeds as follows. The starting point is the canonical DOM Herbrand structure

$\mathcal{HD}_0$ = {bird subcl animal.
bird subcl animal.
penguin subcl bird.
zoo isa zoo.
tweety isa penguin.
lora isa bird.
$\mathcal{A}^*$(attribute,animal) = ((“no”,fly)),
$\mathcal{A}^*$(attribute,bird) = ((“yes”,fly)),
$\mathcal{A}^*$(attribute,penguin) = ((“no”,fly)),
$\mathcal{A}$(attribute,*tweety*) = ((“tweety”, name)),
$\mathcal{A}$(attribute,*lora*) = ((“lora”,name)),
$\mathcal{A}$(child,*zoo*) = ((*tweety*, penguin), (*lora*,bird)),
$\mathcal{A}$(child,*tweety*) = (),
$\mathcal{A}$(child,*lora*) = (),
$\mathcal{A}$(attribute,*zoo*) = (),
$\mathcal{A}$(attribute,*tweety*) = ((“tweety”, name)),
$\mathcal{A}$(attribute,*lora*) = ((“lora”,name)) }

Applying the closure operator yields

$\mathcal{C}\ell(\mathcal{HD}_0) = \mathcal{HD}_0 \cup$ {penguin subcl animal.
tweety isa bird.
tweety isa animal.
lora isa animal. }

with the set of triggers

$$\mathbf{T}(\mathcal{C}\ell(\mathcal{HD}_0)) = \{(\text{tweety isa penguin}, @\text{fly}\bullet\!\!\to \text{“no”}), (\text{penguin subcl bird}, @\text{laying\_eggs}\bullet\!\!\to \text{“yes”})\}\ .$$

Firing the first one and applying the $\mathcal{TX}_P$ operator (which does nothing since there is no program), we get

$$\mathcal{HD}_1 = \mathcal{TX}^{\omega}_{P,\mathcal{C}\ell}(\mathcal{HD}_0 \prec \{\text{tweety}[@\text{fly}\to \text{“no”}]\})$$

i.e., $\mathcal{C}\ell(\mathcal{HD}_0)$ where $\mathcal{A}(\text{attribute}, \textit{tweety})$ is now $((\text{“tweety”}, \text{name}), (\text{“no”}, \text{fly}))$. Then,

$$\mathbf{T}(\mathcal{HD}_1) = \{(\text{penguin subcl bird}, @\text{laying\_eggs}\bullet\!\!\to \text{“yes”})\}\ ,$$

resulting in

$$\mathcal{HD}_2 = \mathcal{TX}^{\omega}_{P,\mathcal{C}\ell}(\mathcal{HD}_1 \prec \{\text{penguin}[@\text{laying\_eggs}\bullet\!\!\to \text{“yes”}]\})$$

with

$$\mathbf{T}(\mathcal{HD}_2) = \{(\text{tweety isa penguin}, @\text{laying\_eggs}\bullet\!\!\to \text{“yes”})\}$$

and

$$\mathcal{HD}_3 = \mathcal{TX}^{\omega}_{P,\mathcal{C}\ell}(\mathcal{HD}_2 \prec \{\text{tweety}[@\text{laying\_eggs}\bullet\!\!\to \text{“yes”}]\})\ .$$

Now $\mathbf{T}(\mathcal{HD}_3) = \emptyset$. The sequence $(\mathcal{HD}_0, \mathcal{HD}_1, \mathcal{HD}_2, \mathcal{HD}_3)$ is a maximal $\mathcal{I}_P$-sequence.
The final structure $\mathcal{HD}_3$ coincides with $\mathcal{C}\ell(\mathcal{HD}_0)$ except

- $\mathcal{A}(\text{attribute}, \textit{tweety})$ is now $((\text{“tweety”}, \text{name}), (\text{“no”}, \text{fly}), (\text{“yes”}, \text{laying\_eggs}))$ and
- $\mathcal{A}^*(\text{attribute}, \textit{penguin})$ is now $((\text{“no”}, \text{fly}), (\text{“yes”}, \text{laying\_eggs}))$.

The result tree does not contain the inheritable properties of classes (since it does not contain the classes at all), but the inherited properties of the instances:

```
<!DOCTYPE mondial SYSTEM "zoo.dtd">
<zoo>
    <penguin name="tweety" fly="no" laying_eggs="yes"/>
    <bird name="lora" fly="yes" laying_eggs="yes"/>
</zoo>
```

The proposal is continued in Section 10.4 with considerations on refining element types, structural inheritance, and the representation of such element types in XML instances.

10 SIGNATURES IN XPATH-LOGIC

Signature information in XPathLog is used both as a *data dictionary* in the SQL-style which is uniformly accessible and extensible by XPathLog rules, and for defining views on the internal database as projections.

XPath-Logic seamlessly integrates signature information via *signature atoms*: For every element type ("class"), a signature is specified which gives the result types of its properties (subelements and attributes). Thus, XPathLog's signature is not a complex type specification as DTDs, XML Schema, or the types of the XML Query Algebra, but a "lightweight" formalism, influenced by the experiences with F-Logic. There are no complex type definitions, it does even neither specify order nor cardinalities.

Following the object-oriented tradition, the notion of *classes* is used for modeling and classifying objects (isa-hierarchy), whereas the notion of *types* covers the more formal, structural aspects. In this work, the class hierarchy and the type hierarchy coincide. In this chapter, the notion of classes is used, emphasizing the modeling aspect of signatures.

10.1 Representation of Signatures

The representation of signatures by *signature atoms* is only concerned with the *names* in X-structures as defined in Definition 5.3 using

- $\mathcal{N}_{\mathcal{E}}$: element names,
- $\mathcal{N}_{\mathcal{A}}$: attribute names,
- $\mathcal{N}_{\mathcal{C}}$: partitioned into $\mathcal{N}_{\mathcal{CE}}$ and $\mathcal{N}_{\mathcal{CL}}$ for element and literal classes (cf. Definition 9.1).

Using the DTD metadata model, $\mathcal{N}_{\mathcal{E}}$ and $\mathcal{N}_{\mathcal{CE}}$ coincide.

Definition 10.1 (X-Structures with Signatures)
For including signatures, X-structures are again extended with two more (partial) mappings:

$$\mathcal{E}^{\Rightarrow} \;:\; \mathcal{N}_{\mathcal{CE}} \times \mathcal{N}_{\mathcal{E}} \to 2^{\mathcal{N}_{\mathcal{C}}} \quad \text{and} \quad \mathcal{A}^{\Rightarrow} \;:\; \mathcal{N}_{\mathcal{CE}} \times \mathcal{N}_{\mathcal{A}} \to 2^{\mathcal{N}_{\mathcal{C}}}$$

which specify the result types of applying a property to an object of some type/class: $\{cl_1, \ldots, cl_n\} \subseteq \mathcal{E}^{\Rightarrow}(cl, el)$ denotes that subelements with "name" el of elements of class cl are (conjunctively) of types/classes $cl_1, \ldots, cl_n$; analogously for attributes.

Similar to Definition 9.5, the monotonic (structural) inheritance of signatures adds another closure requirement (propagation to subclasses):

- if $c \in \mathcal{E}^{\Rightarrow}(class, attr)$ and $class \in \mathcal{SC}(sub)$, then $c \in \mathcal{E}^{\Rightarrow}(sub, attr)$, and
- if $c \in \mathcal{A}^{\Rightarrow}(class, attr)$ and $class \in \mathcal{SC}(sub)$, then $c \in \mathcal{A}^{\Rightarrow}(sub, attr)$. □

Again, the extension of these mappings is not contributed by the XML instance, but by the DTD (or by the XML Schema definition, see Section 10.2.2).

Definition 10.2 (XPath-Logic Signature Atoms)
XPath-Logic is extended for handling signature knowledge as follows:

- *XPath-Logic signature atoms* are of the form

  $class_1[subelement{\Rightarrow}class_2]$ or $class_1[@attribute{\Rightarrow}class_2]$

  for $class_2 \in \mathcal{E}^{\Rightarrow}(class_1, subelement)$, or $class_2 \in \mathcal{A}^{\Rightarrow}(class_1, attribute)$.

- Similar to $\mathcal{A}^*$ in the previous section, the semantics $\models$ and $\mathcal{QB}$ of these atoms are defined straightforwardly based on evaluating terms and checking $\mathcal{E}^{\Rightarrow}$, $\mathcal{A}^{\Rightarrow}$, or $\mathcal{HD}$, respectively. □

Remark 10.1
The above signatures mirror the differences between the *edge-labeled* XPath-Logic data model, the basic XML tree model, and the data model used in XML Schema:

- Attributes: for non-reference attributes, the result class is always literal or a subclass of it (number, string).
  For reference attributes (which are resolved automatically when parsing an XML document), the result class is always object or a subclass of it (i.e., element classes or user-defined classes).

- Subelements: When using a DTD, $\mathcal{E}^{\Rightarrow}(el, subelement) = subelement$ since element names and element types are not distinguished. When using an XML Schema metadata description, the name of the subelement relationship in general differs from the name of the subelement type. The same may happen when executing updates and linking an element of original type *type* as $subelement \neq type$-subelements. □

Example 10.1 (Overlapping Trees)
Consider again an excerpt of the running example. For

```
result isa metropoles.        <!-- (document) type of result -->
result[metropole→C] :- cia/city→C[population > 1000000].
```

(regarding population as an annotated literal *(see Section 5.6, and Example 3.3 for the same in XPath)), the result signature is specified by the atom*

```
metropoles[metropole⇒city].
city[name⇒name].                name[text()⇒string].
city[population⇒population].    population[text()⇒numeric].    population[@year⇒numeric].
```

Then, e.g., berlin *originally is a* city *element in the mondial tree, but occurs as a* metropole *element in the result tree (cf. Figure 10.1). Note that wrt. the* result *tree, the reference attribute* berlin/@country *is a dangling reference.*

10.2 Deriving the Signature

Often, XML documents provide a metadata description in form of a DTD or an XML Schema instance which both provide a much more detailed description than needed for signature atoms. The signature atoms can be derived from these sources.

10.2.1 DTD

DTDs (see Section 2.2) are not directly accessible by most XML tools since they are not in XML syntax, and there is no natural translation into the XML data model. Note that even the DOM model does not allow for accessing metadata about a stored document (which has been used when

```
<metropoles>
  <metropole id="city-berlin"
             country="D">
    <name>Berlin</name>
    <population year="95">
      3472009
    </population>
  </metropole>
</metropoles>
```

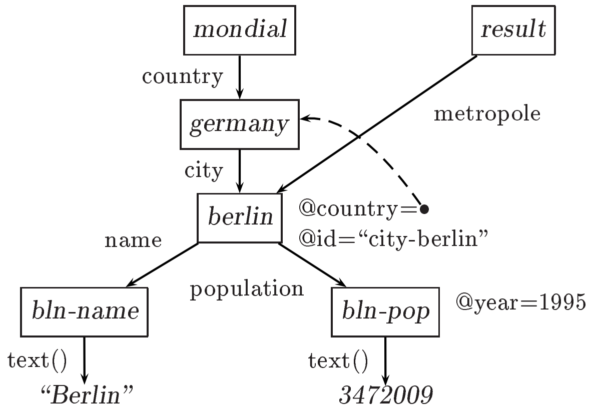

Figure 10.1: Element with multiple parents

parsing the document). It is even not possible to check which attributes are declared as `ID` [Beh01]. Validating XML parsers *implicitly* parse the DTD for obtaining information on the attribute types (for resolving `ID/IDREF` attributes). Commercial tools allow to access the DTD as a metadata specification via their user interfaces.

Using the LoPiX system (see Section 15), a DTD accessible by an url *url* can be transformed into signature atoms using the built-in active Web access method *url*.parse@(dtd) (see Section 15.3). The mapping is defined as follows:

Element declarations: For an element declaration

<!ELEMENT *element contentsmodel*> ,

- object $\in \mathcal{SC}$(*element*), and
- for every elementtype *type* occurring in *contentsmodel*, *type* $\in \mathcal{E}^{\Rightarrow}$(*element*,*type*) , and
- if *contentsmodel* is either `(#PCDATA)` or mixed as `(#PCDATA|...)`, then literal $\in \mathcal{E}^{\Rightarrow}$(*element*,text())

Attribute declarations: For an attribute declaration

<!ATTLIST	*element*	*attribute*$_1$	*attr_type*$_1$	*attr_constr*$_1$
		:	:	:
		attribute$_n$	*attr_type*$_n$	*attr_constr*$_n$>

- literal $\in \mathcal{A}^{\Rightarrow}$(*element*,*attribute*$_i$) if *attr_type*$_i$ is `CDATA`, `NMTOKEN`, `NMTOKENS`, or `ID` ,
- object $\in \mathcal{A}^{\Rightarrow}$(*element*, *attribute*$_i$) if *attr_type*$_i$ is `IDREF` or `IDREFS` ,
- *new_type* $\in \mathcal{A}^{\Rightarrow}$(*element*, *attribute*$_i$) if *attr_type*$_i$ is an enumeration. In this case, where *new_type* is defined to contain all values (as constants, not as strings) of the enumeration (see Example 10.2).

Since DTDs do not contain any information about the target type of `IDREF` attributes, the result type is always only known to be object. Similarly, `CDATA` and `NMTOKENS` are only known to result in type literal.

Note again that cardinalities and order are not considered in XPathLog signature atoms.

Example 10.2 (Enumeration types)
The representation of rivers in MONDIAL *is given in the DTD as follows:*

```
<!ELEMENT river (name,to?,located*,length?)>
<!ATTLIST river id ID #REQUIRED
                country IDREFS #REQUIRED>
<!ELEMENT to EMPTY>
<!ATTLIST to watertype (river|sea|lake) #REQUIRED
              water IDREF #REQUIRED>
```

where river, sea, *and* lake *are the elementtypes describing waters. An instance is, e.g.,*

```
<river is="river-rhein" country="CH A FL D F NL">
   <name>Rhein</name>
   <to watertype="sea" water="North Sea">
</river>
```

Initially, the new class has only an internal name, and the three watertypes are members of it:

```
"mondial-2.0.dtd".parse@(dtd).    % parses the DTD
?- to[@M⇒V].

M/watertype    V/f0_143
M/water    V/object

?- X isa f0_143.

X/sea
X/river
X/lake
```

When parsing the corresponding XML instance, the attribute values are set appropriately:

```
"mondial-2.0.xml".parse@(xml).    % parses the XML instance
?- //river[name/text() = "Rhein" and
        to[@watertype→WT1 and @water[name/text()→N] isa WT2]].
WT1/sea    N/"North Sea"    WT2/sea
```

In XML/XPath, there is no such direct connection between element names and literals. For an example dealing with this aspect, see Example 14.4.

Remark 10.2
The above design decision as implemented in LoPiX can be criticized: an enumeration may also constrain the range of a string-valued attribute, e.g., for the state codes of a US database:

```
<!ELEMENT city (...)>
<!ATTLIST city state (AL|AK|AR|AZ|CA|...) #REQUIRED
               ...     >
```

A refined implementation could use the following heuristics: if the items of the enumerations are *names* occurring the the DTD (as in the above example, *sea*, *river*, and *lake*), the enumeration items are interpreted as objects, whereas in the other case, they are simply values. □

Example 10.3 (Mondial Signature and DTD)
The complete DTD mondial-2.0.dtd (describing a hierarchically structured document type which contains examples for most XML constructions) can be found in Appendix A. The signature which is extracted by the above mapping is similar to the one given in Appendix C (which is extracted from the XML Schema). An important difference is that the DTD does not allow for local type definitions. In Mondial, *this concerns e.g. the* population *elements (cf. Example 3.16). Here, the DTD and the derived signature contain only the global, general type:*

country [name⇒name and population⇒population and ...].
city [name⇒name and population⇒population and ...].
population[text()⇒literal and @year⇒literal].

For the more detailed metadata information about MONDIAL *in XML Schema, see the subsequent section, especially Example 10.4.*

10.2.2 XML Schema

XML Schema [XML99a] (see also Section 3.7) instances are valid XML instances. Thus, they can be queried in XPathLog. XML Schema distinguishes *element types* (which define a *type* in the sense of *datatypes*) from *element names* (which occur in the tags of the ASCII representation, and which are used for navigation).

XML Schema documents are translated as follows into signature atoms (the rules are *not* exactly those used in the actual program given in Appendix C), but a stripped version which illustrate the core procedure. The actual program introduces a constant for every datatype):

- the *primitive datatypes* of XML Schema are subclasses of literal, note that multivalued *simpleTypes* (e.g., `NMTOKENS`) are split on the data level.
- types defined as *simpleTypes* also become subclasses of literal:

  ST subcl literal :- //simpleType→ST.

- types defined as *complexTypes* become subclasses of object:

  CT subcl object :- //complexType→CT.

- <element> and <attribute> declarations define properties (yet, of nothing, simply properties) by defining a name and a result data type.

  E.g., the global attribute definition

  <attribute name="area" type="float"/>

  is (virtually) translated into [@area⇒float] – which is not yet an XPathLog signature atom, but will become one when used by a host class.

- XML Schema *complexTypes*: As already stated in Section 3.7, the base and derivedBy relationships induce a hierarchy on types which is incorporated into the class hierarchy, i.e., a *complexType* becomes a subclass of the type from which it is derived.
  - *ComplexTypes* can be derived from a *simpleType* (which then defines the text contents):

    CT subcl ST, CT[text()⇒ST] :- //complexType→CT[base→ST], ST isa simpleType.

    (see example below for the consequences on the model, leading to *annotated literals*, cf. Section 5.6.)
  - *ComplexTypes* can be derived from a *complexType* (which then defines contents and attributes) by extension and/or refinement. The signature of the type which is used also applies to the derived type (*structural inheritance*, see also Section 10.3):

    (*i*) CT subcl BT:- //complexType→CT[base→BT], BT isa complexType.
    (*ii*) CT[S⇒TT] :- //complexType→CT[@base→BT], BT isa complexType, BT[S⇒TT].
    (*iii*) CT[@A⇒AT] :- //complexType→CT[@base→BT], BT isa complexType, BT[@A⇒AT].
    (*iv*) CT[@A•→V] :- //complexType→CT[@base→BT], BT isa complexType, BT[@A•→V].
    (*v*) CT[@A••→V] :- //complexType→CT[@base→BT], BT isa complexType, BT[@A••→V].

    Here, (*ii*) and (*iii*) are *closure properties* which are not necessarily materialized in an implementation. The handling of (*iv*) and (*v*) has been described in Section 9.2.

– extensions to the contents model and the attributes are specified by referring to properties (or defining them locally) and specifying a cardinality and optionally a default or fixed value of the property:

  CT[E⇒T] :- //complexType→CT//element/@ref[@name→E and @type→T].

– define additional content model particles (appended to the content model of the type which is extended). Since cardinalities and order are not considered in XPathLog signature atoms, only the names of subelements and their types are queried:

  CT[E⇒T] :- //complexType→CT//element[@name→E and @type→T].

– define additional attributes:

  CT[@A⇒T] :- //complexType→CT//attribute/@ref[@name→E and @type→T].
  CT[@A⇒T] :- //complexType→CT//attribute[@name→E and @type→T].
  CT[@A•→V] :- //complexType→CT//attribute/@ref[@use→"default" and @value→V].
  CT[@A•→V] :- //complexType→CT//attribute[@use→"default" and @value→V].
  CT[@A••→V] :- //complexType→CT//attribute/@ref[@use→"fixed" and @value→V].
  CT[@A••→V] :- //complexType→CT//attribute[@use→"fixed" and @value→V].

The complete program is given in Appendix C, handling types by introducing constants for identifying classes instead of associating the signature directly with the nodes of the XML Schema instance (as would be done if using the above rules). The complete derived signature can also be found in Appendix C.

The class hierarchy induced by XML Schema specifications also uses the notion of *annotated literals* which have been investigated in Section 5.6:

Example 10.4 (Annotated Literals: Signature and Instances)
Consider again the fragment of the Mondial *XML Schema description given in Example 3.16 which different population element types for countries and cities:*

```
<complexType name="country">
  <attribute name="car_code" ... />
  <element name="population" type="integer" .../>
    ⋮
</complexType name="country">

<complexType name="city">
  <element ref="name"/>
    ⋮
  <element name="population" ...>
    <complexType base="integer" derivedBy="extension">
      <attribute name="year" type ="date" use="optional"/>
    </complexType>
  </element>
    ⋮
  </complexType>
```

Both the local types city:population *and* country:population *are subclasses of a general type* population and *of the literal type* integer. *Thus, their instances are both objects (as elements) and literals (as values). (Note that it is even possible to define a global type* population *which then refined by* city.population *by providing an additional* year *attribute.)*
The corresponding XPathLog signature atoms are

country[name⇒name and population⇒country:population and @car_code⇒string].

city[name⇒name and population⇒city:population and ...].

```
country:population subcl literal.
country:population subcl object.
country:population subcl integer.
country:population [text()⇒integer].

city:population subcl object.
city:population subcl literal.
city:population subcl integer.
city:population [text()⇒integer and @year⇒date].
```

As a consequence of both population *types being subclasses of* literal *and* object*, instances of these types are expected to act as integers, and as complex objects. It is possible to compare them with integer values*

//country[population > 5000000]/name/text()

and to query the year *attribute of* country.population*, e.g.,*

//city[population[@year < 1990] > 5000000]/name/text() .

as annotated literals *(cf. 5.6).*

10.3 Structural Inheritance

In presence of a class hierarchy, the *structural information* of superclasses is inherited to subclasses. For the lightweight schema information (instead of strict typing), this means that signature atoms which hold for a class also hold for its subclasses. Structural inheritance is – in contrast to behavioral (value) inheritance – monotonic: The semantics of redefining a signature for a subclass is not *overriding*, but *refining* (recall that multiple signature atoms $class[property{\Rightarrow}class_1]$ and $class[property{\Rightarrow}class_2]$ are interpreted conjunctively in Definition 10.2).
Structural inheritance of signature atoms is added to the closure operator:

Definition 10.3 (Closure Axioms: Extended $\mathcal{TX}_P$-Operator)
Further extending Definition 9.7, for a DOM Herbrand Structure $\mathcal{HD}$, the *closure* operator, $\mathcal{C}\ell(\mathcal{HD})$ is extended by

- if c subcl $d \in \mathcal{HD}$ and $d[e{\Rightarrow}f] \in \mathcal{HD}$ then add $c[e{\Rightarrow}f]$ to $\mathcal{HD}$,
- if c subcl $d \in \mathcal{HD}$ and $d[@a{\Rightarrow}f] \in \mathcal{HD}$ then add $c[@a{\Rightarrow}f]$ to $\mathcal{HD}$.

Again, the $\mathcal{TX}_{P,\mathcal{C}\ell}$-Operator defines the semantics of a program. □

Recall that the $\mathcal{C}\ell$ operator is not necessarily implemented by materializing the derived atoms.

Example 10.5 (Signature: Structural Inheritance)
Consider again the zoo *DTD given in Example "zoo.dtd", inducing the following signature:*

```
zoo[animal⇒animal].
animal[name⇒literal].
animal[fly⇒literal].
bird[fly⇒literal].
bird[laying_eggs⇒literal].
penguin[fly⇒literal].
bird subcl animal.
penguin subcl bird.
```

Here, the closure adds the following atoms by monotonic, structural inheritance:

bird[name⇒literal].
penguin[name⇒literal].
penguin[laying_eggs⇒literal].

10.4 Proposal: Structural Refinement and Inheritance in XML

Since (i) the XML data model does not support a class hierarchy, and (ii) the definition of a type definition hierarchy on metadata level in DTDs and XML Schema is not completely satisfying (see below), a proposal for extending the XML model with classes has been developed (extending the proposal described in Section 9.2.2 for adding class information to DTDs).

10.4.1 Structural Inheritance

When using a *type hierarchy*, subtypes inherit their structure from their supertypes (depending on the model, it may be allowed to have several lowest supertypes), called *structural inheritance* (the proposal described in Section 9.2.2 describes how to extend DTDs to specify a class hierarchy).

Since DTDs don't know a type hierarchy, there is also no notion of structural inheritance. The XML Schema working draft includes structural inheritance with the definitions of *simpleTypes* and *complexTypes* (cf. Section 10.2.2):

- derived *simpleTypes* inherit all constraints (expressed via facets) from the supertype,
- the inheritance step from a *simpleType* supertype to a *complexType* subtype is more involved, inheriting the supertype's constraints to the *text contents* of the subtype (see Section 5.6). The same holds when an *element declaration* directly uses a *simpleType*.
- if the supertype (and then also the subtype) is a *complexType*, the subtype inherits all attribute declarations (and may add its own ones) and inherits the contents model of the supertype.

Here, especially inheriting a contents model has to be investigated. In XML Schema, the subtype's contents model is obtained by *adding the additional contents model specifications of the subtype at the end of the contents model specification of the supertype* [XML99b, Section 2.2.1.3] which is an at least questionable approach:

- a *refinement* of the contents model in the sense of stronger structure requirements, or requiring finer types for some properties is not possible.
- similar, an overwriting of attribute specifications is not possible.
- the effect of "redefining" an element or attribute declaration which has already been defined for a supertype is undefined in [XML99b].

Thus, the type definition hierarchy of XML Schema is not compatible to the notion of type inheritance in object-oriented models such as ODMG [CB00].

Proposal for a type hierarchy in XML. For defining a notion of subtyping wrt. the above aspects, the formalism in the DTD is more suitable: when a subtype is defined, either

- no contents model is given. Then, the subtype uses the same contents model as the supertype.
- a refining contentsmodel is given, i.e., which (i) strengthens the structural requirements or (ii) uses subtypes of the element types requires in the supertype contents model at some positions.

10.4.2 Representation of Class Membership in XML Instances

Having a detailed class hierarchy as known from object-oriented models, the detailed modeling has also to be implemented by the basic XML concepts such as the DOM and XML Query data model, the DTD and XML Schema metadata specifications, the ASCII representation of XML, and XPath.

Consider the following situation: the concept *person* has numerous specialized, in general overlapping subclasses: *john* may be a person, having a name, a birth date, a social security number, an address (or more addresses wrt. given dates). As a person, he also has father and mother relationships and perhaps children relationships. Additionally, he is male which presumably adds some information on military service etc. He is also a professor, having an office, an education, some projects etc. Perhaps he is also an amateur musician, which defines additional properties.

The relevant aspect now is that every such class defines a signature describing the properties of a person from the point of view of this individual subclass. From the XML point of view, every class corresponds to an element type which defines a contents model. At the end, objects which belong to many classes do not satisfy the contents model of any of these classes, but the "union" of all its contents models. As a consequence, it is hard to represent such objects (e.g., *john*) in order to be valid to any DTD. There are two possibilities:

- Collecting: There is one element, representing *john* with all his properties. When *john* is seen as a professor, there is a reference to this element. The object does not satisfy the contents model of any of the classes where it belongs to, but it satisfy a kind of "union" of all their contents models.
- Distributed: *john* is distributed over several elements. There is a //persons/person element, a //university/staff/professor element, a //culture/concert/musician element etc. All of them represent *john*. How to search for a professor who is able to play piano and less than 50 years old?

Collecting. In object-oriented frameworks, an object which belongs to multiple classes is *casted* to the class which is currently needed: e.g., (professor)*john*.project addresses his *project* members. The proposal for XML is the same, casting the properties:

```
<persons>
   <person id=“person-1234” name=“John” birthday=“31.2.1950” ...
        (professor)affiliation=“univ-1234” (professor)room=“...” ...
        (musician)amateur=“yes”>
      <address > ... </address>
          <? xclasses cast role=“professor”?>    <!– XML processing instruction –>
      <project> ... </project>
      <project> ... </project>
          <? xclasses cast=“musician”?>
      <performance concert=“concert-4321” ...> ... </performance>
      <performance ...> ... </performance>
      ⋮
</persons<

<university>
   <staff professor=“person-1234 ...”>
          ⋮
   </staff>
   ⋮
</university>
```

```
<culture>
    <concert id="concert-4321" date="1.1.2002"
             place="Town Hall" musician="[person-1234] ...">
             ⋮
    </concert>
    ⋮
</culture>
```

The processing instructions tell an xclasses-aware application that the following properties apply to *john* as an instance of a specialized subclass.
Querying is as usual in XML and XPath, e.g.,

```
//university/staff/@professor[name→N and course="CS9876"]/project
```

Queries (although neither XPath nor XPathLog are type-checking) may also cast navigation steps, e.g., the query

```
//person[age()>50 and @name→N and (professor)@affilitation]/
        (musician)performance/@concert[place→"Town Hall"]/date→D.
```

checks if there is a person over 50 who is a professor at some institute and gives a concert in the town hall, and outputs the name and the date.

For checking validity of the XML instance, every element is projected to the subclasses where it it casted to, and verified wrt. the contents model and attribute list of the subclass.

The above mechanism has some similarities with the *namespace* concept; in some sense it is a continuation in the direction of data integration: namespaces allow for describing a concept from different points of view. In course of data integration, instances from different sources have to be handled in a single instance. Here a solution is to *fuse* their instantiations, resuting in an element which collects the properties of both sources (cf. Section 11.6). On the signature level, the resulting element is of an element type which combines the structural definitions of both types.

Distributed. Here, the elements which represent an object in its different roles/subclasses must be connected in some way. Using the same id or key (XML Schema), it is left to the implementation how to handle this situation internally (e.g., using an equational theory, or mapping it to the above collecting model).
An XML instance looks as follows:

```
<persons>
    <person [id="person-1234"] name="John" birthday="31.2.1950" ...>
        <address > ... </address>
    </person>
        ⋮
</persons>

<university>
    <staff>
        <professor [id="person-1234"] room="...">
            <project> ... </project>
            <project> ... </project>
        </professor>
            ⋮
    </staff>
```

```
      ⋮
</university>
<culture>
  <concert date="1.1.2002">
    <place> ... </place>
    <musician [id="person-1234"] amateur="yes">
      <instrument> ... </instrument>
    </musician>
      ⋮
  </concert>
    ⋮
</culture>
```

Validation of the document proceeds as usual: each element instance describes only the properties which are required in the current context. In contrast to the XML model, an `ID` now does not identify a *unique* element, but is used for *identifying* several elements.

Querying is also as usual, but the *implementation* of navigation is more involved since it must be searched which of the "element fragments" contains information relevant to the query.

Concerning an XPathLog implementation, the collecting model is recommended. Then, only the export functionality for generating the ASCII representation must be tailored to the "external" model.

11 DATA INTEGRATION: HANDLING MULTIPLE XML TREES

As a consequence of the conclusions drawn in Section 4, the order of subelements is in general not relevant for information integration aspects from the database point of view. Thus, in this section, the order is not dealt with, but the approach can also be applied to the ordered model.

As described in Definition 5.4, XPath-Logic and XPathLog are not based on a tree model, but on X-structures which induce a *navigation graph* (cf. Figure 5.1). In contrast to the *node-labeled* DOM and XML Query data models, the navigation graph is *edge-labeled*, i.e., the names of the subelements and attributes are annotated to the *edges*, not to the nodes. This allows for defining overlapping tree views where the same elements may occur under different names, as illustrated in Example 10.1. For another example consider Examples 7.4 and 7.5 where the cities *Munich* and *Nürnberg* are *city* subelements of *Germany* and of *Bavaria*.

Thus, the navigation graph is not intended to represent "the XML document", but a *database* where XML documents may be defined as "views" – some of the views are the original XML trees. In general, an X-structure may represent several trees which even may be overlapping, i.e., subtrees can belong to several trees.

This data model covers the XML data model: for the *canonical X-structure* (cf. Definition 5.5) to an XML instance, the navigation graph is a tree.

The above-mentioned examples just illustrated the feature of having multiple parents by small examples. In the following, we focus systematically on the modeling aspect of multiple trees for *data restructuring* and *integration*. The presented strategies show the flexibility of the XPath-Logic data model. The strategies are then applied in the case study described in Section 12.

11.1 Projection By Signature

Usually, database languages define the output as a projection of an intermediate result (or of the database), e.g., the SQL `SELECT` clause:

```
SELECT name, area, population
FROM country;
```

Projection in XML trees is more involved since not only the name of a property, but also the tree structure must be considered. Quilt and XQuery (see Sections 3.10 and 3.11) provide the `FILTER` operator for applying a projection on a tree. This operator is not exactly a projection, but a projection-and-omitting.

In XPathLog, a projection of a tree is specified by giving a root node and a *signature* as defined in Section 10. Starting from the root node, the navigation graph is traversed by following all links which are covered by the given signature.

Definition 11.1 (XML Tree View)
For an XML database $\mathcal{X}$, a set Sig of signature atoms, and a node r, the *tree view rooted in* r, $\mathcal{Y} := \mathcal{X}(r, Sig)$ is defined as the graph

- $r \in \mathcal{V}_\mathcal{Y}$,
- if $n \in \mathcal{V}_\mathcal{Y}$ and n isa c, then
  - for all signature atoms $c[e{\Rightarrow}c'] \in Sig$: if $\mathcal{E}(n,e,i) = v$ for some i and v, the edge $\mathcal{E}(n,e,i)$ belongs to $\mathcal{Y}$ – and $v \in \mathcal{V}_\mathcal{Y}$, and
  - for all signature atoms $c[@a{\Rightarrow}c'] \in Sig$, $\mathcal{A}(n,a)$ belongs to $\mathcal{Y}$.
- for all $v \in \mathcal{V}_\mathcal{Y}$, $\mathcal{C}_\mathcal{Y}(v) = \{c \in \mathcal{C}(v) \mid \text{there is a signature atom } c[\ldots] \in Sig\}$ (collecting the relevant classes only). □

Note that the result view may contain dangling references to nodes which are not part of the tree (cf. Section 11.8).

In contrast to the Quilt/XQuery `FILTER` operator, projection by signature does not *change* the tree structure, but completely prunes subtrees and attributes. Changing the tree structure by omitting intermediate elements can be done by adding direct links.

Example 11.1 (Mapping Quilt Filtering to XPathLog Tree Views)
The Quilt filter given in Example 3.26 is expressed in XPathLog as follows:

```
<!– bridge intermediate elements –>
C[city→Cty] :- //country→C[descendant::city→Cty].
<!– move names into text contents –>
C[text()→N] :- //country→C[name/text()→N].
C[text()→N] :- //city→C[name/text()→N].
<!– specify signature –>
mondial[country⇒country].
country[@car_code⇒string].
country[text()⇒string].
country[city⇒city].
city[text()⇒string].
```

Exporting the subtree rooted in mondial *yields the same output as given in Example 3.26.*

11.2 Generating an Isolated Result Tree

The projection described in the previous paragraph involved only one tree in the database from which a "result" tree was defined as a projection. Projection is only suitable if a property should be handled homogeneously for all elements in the tree. The "opposite" strategy is to create the result tree completely from scratch by collecting nodes (in the extreme case, only literals, i.e. attribute values and text contents) and structuring them by creating new elements and subelement relationships.

Example 11.2 (Isolated Result Tree)
The following program creates a result tree consisting of all organizations which have at least one member with a gross domestic product per persona of more than 1000$. Each organization contains all members as subelements with name, gdp, population, and gdp/persona (gdpp) subelements.
The first rule,

```
result/organization[@abbrev→A] :-
  //organization[abbrev/text()→A]/members/@country
        [gdp_total/text()→_G and population/text()→_P],
    _X = _G * 1000000 div _P, _X > 1000.
```

creates new *organization elements (by the path expression* result/organization[...]*) in the result tree and sets their* abbrev *attribute. The second rule then creates* new *country subelements of these organizations:*

```
O/country[name→T and gdp→G and gdpp→X] :-
    result/organization→O[@abbrev→A],
    //organization[abbrev/text()→A]/members/@country
          [name/text()→T and gdp_total/text()→G and population/text()→P],
    X = G * 1000000 div P, X > 1000.
```

The result projection is defined by the following signature

```
result[organization⇒organization].
organization[@abbrev⇒string].
organization[country⇒country].

name subcl textonly.
gdp_total subcl textonly.
gdpp subcl textonly.
textonly[text()⇒string].

country[name⇒textonly].
country[gdp_total⇒textonly].
country[gdpp⇒textonly].
```

Building a separate tree is the recommended strategy if the structure of the result is very different from the original tree.

11.3 Generating a Result Tree by Selecting and Linking Subtrees

As a compromise between the above "extreme" strategies, a result tree can be created by *linking* subtrees of the original document to it, extending them appropriately, and projecting the result.

Example 11.3 (Linked Result Tree View)
Consider again the task given in Example 11.2. The result tree can also be generated by linking*:*

- *the appropriate organization elements become children of the* result *node,*
- *the appropriate countries become subelements of the organizations and are extended by* gdpp*.*

The result tree is constructed by the following program:

```
C[gdpp→X] :-
   //country→C[gdp_total/text()→G and population/text()→P],
   X = G * 1000000 div P, X > 1000.

result[organization→O[@abbrev→A and country→C]] :-
   //organization→O[abbrev/text()→A]/members/@country→C[gdpp].
```

The output signature is the same as above.
Note that in the whole database, there exists only one country *element for each country. On the other hand, the original tree is also changed (the* gdpp *subelements also belong to the original tree):*

```
?- //country[name/text()→N]/gdpp→X.
```

yields the gdp per persona for all countries such that gdpp>1000*.*

The above strategy needs much less storage than generating a completely new tree.

11.4 Namespaced Input and Multiple Input Trees

As described in Section 2.3, *namespaces* can be used for distinguishing elements originating from different sources. Especially for data integration, namespaces are an important feature allowing to distinguish different source trees (even when elements of different trees are *fused* (see below) during integration) and also possibly generating several result trees as views on the internal database.

When integrating multiple input trees, namespaces are associated with *groups* of documents. Documents which semantically belong together and use the same DTD also share the same namespace. The actual decision depends on the situation, e.g., if the task consists of combining consistent sources describing different but overlapping application domains (e.g., a flight database and hotel bookings), or combining sources containing possible inconsistencies on the same application domain (e.g., integrating catalogs from different suppliers).

Note that even if two sources use the same ontology, but represent different views, e.g., different timepoints, it is recommended to use different namespaces.

Example 11.4 (Namespaced Input)
Consider the MONDIAL *database distributed over the following sources:*

1. *CIA countries: a source which contains all countries*
2. *CIA organizations: a source which contains all organizations*
3. *GlobalStatistics: a separate country source for each country*

Then, e.g., the following namespaces can be used:

1. cia: *for the first page,*
2. orgs: *for the second page,*
3. gs: *for all Global Statistics pages.*

Note that then, about 250 country trees exist using the gs: *namespace – since they also use a common DTD and describe the same properties.*

The edge-labeled navigation graph model allows to combine the namespace concept with multiple *overlapping* trees.

The integration process starts with parsing all source trees augmented with suitable namespaces. Then, the result tree is constructed based on nodes and literals of these trees by the following operations which are described in the sequel:

1. Synonyms: identifying and renaming properties,
2. Element fusion: identifying elements in different sources which represent the same object in the application domain,
3. Linking and Collecting: elements and tree fragments are linked together to define a result tree view.

11.5 Synonyms

In contrast to the pure DOM model, the names are also elements of the universe which can be bound to variables, used in predicates, and especially *equated* with other names and synonyms. The latter proves very useful in data integration: When elements from a source are integrated into the result tree, in general also some of their properties should completely become properties of the result view. Here, synonyms are an efficient means for taking a whole property from a source tree (and namespace) to the result tree: By equating *namespace:name*$_1$ = *name*$_2$.

the property *name*$_2$ is defined to have the same extension as the original property *namespace:name*$_1$. E.g.,

```
cia:name = name.
cia:area = area.
cia:population = population.
```

defines the name, area, and population properties of countries of the result view. This does *not* introduce new children or attribute nodes, but "only" defines alternative access paths.

Using this strategy, the internal database can be seen as a "two-level" model: the source(s) provide tree structures which *may* be used. The result tree is then defined using parts of this structure, and extending it.

Example 11.5 (Namespaced Input and Synonyms)
Consider once again the task given in Example 11.2. Now, we associate the original XML tree with the "cia:" namespace (cf. Section 15.3 for the implementation in LoPiX*). Then, the result tree is defined as a projection and refinement from the namespaced tree, mainly using nodes which are already present in the source tree:*

- *from the countries, the* name *and* gdp *subelements are used (introducing the synonyms* gdp_total *and* name *for* cia:gdp_total *and* cia:name*); also the* cia:text() *relationship is accessible by* text()*.*
- *the* gdpp *subelement is added for the relevant countries,*
- *the result tree structure is defined by appropriately linking some of the* organization *and* country *elements.*

```
<!– input: the source tree uses the namespace "cia:" –>

gdp_total = cia:gdp_total.
name = cia:name.
text() = cia:text().

C[gdpp→X] :-
    //cia:country→C[gdp_total/text()→G and cia:population/text()→P],
    X = G * 1000000 div P, X > 1000.

result[organization→O[@abbrev→A and country→C]] :-
    //cia:organization→O[cia:abbrev/text()→A]/cia:members/@cia:country→C[gdpp].
```

The projection of the result *tree with the signature given in Example 11.2 yields the desired view.*

In the above example, the unique original tree was equipped with a namespace to allow for a projection by synonyms from the original tree. In the following, the strategy is applied to multiple input trees.

11.6 Fusing Elements and Subtrees

When integrating data from several sources, there are often elements in different sources which represent the same real-world entity, e.g., countries in CIA and in GlobalStatistics. In the resulting database, the information of these elements should be collected in a single element. With XPathLog, such elements can be *fused* into an element which

1. is still an element of *both* source trees, i.e., positive queries against the original tree using the original namespace still yield at least the original answers,
2. collects the attributes of both original elements,
3. collects the subelements of both original elements.

(1) is easily satisfiable in XPathLog by adding appropriate links to the navigation graph. (2) does also not cause any problems: if the original elements use different namespaces, the attributes are simply collected, otherwise for the attributes which are present for both original elements, their values are accumulated. Attributes are not ordered, and also the values of multivalued attributes are not ordered. Only (3) needs a further specification, how the order of subelements is dealt with. Regarding the problem from the database point of view, this aspect can be ignored, accepting any kind of union of two lists (cf. Section 4). (As an extension, if a DTD or XML Schema description is given which uniquely defines how to order the children, this may be used).

Example 11.6 (Integration: Object Fusion)
See the description of the MONDIAL *data sources in Section 12 and excerpts of the trees with namespaces as shown in Figure 11.1. Both sources,* cia *and* gs*, describe countries: where* cia *contains information about name, area, population, capital, and languages, and* gs *contains information about cities. An obvious and typical integration step is to unify the countries in the* cia *tree with the countries in the* gs *tree and link them to the result tree root node:*

```
result[country→C1], C1 = C2 :-
        cia/cia:country→C1[@cia:name→N], gs/gs:country→C2[@gs:name→N].
```

The above rule makes the fused country a child of the result *node: For every country c which is present in both databases (there are some* cia *"countries" which are not actual countries but territories, sometimes even unpopulated), the country elements representing c in* cia *and* gs *are identified and the result is then an element of three trees:* cia*,* gs*, and* result *(cf. Figure 11.2). The example is continued below.*

Often, the identification of corresponding elements is non-trivial, including resolving different ontologies, searching for keys and comparing them, and excluding exceptions and conflicts. Additionally, entities which are represented in only one of the sources probably must be collected.

The resulting elements are linked to the result tree, and the properties which should be contained in the result view are defined. This can be done either by introducing a synonym in the result namespace for a property in one of the source namespaces, or by adding appropriate links parallel to the already existing link (if a property is copied only for some of its instances).

Example 11.7 (Integration: Synonyms)
Consider the situation obtained in Example 11.6. After defining the synonyms

```
cia:name = name.                  gs:city = city.
cia:area = area.                  gs:name = name.
cia:population = population.      gs:population = population.
cia:language = language           gs:text() = text().
```

the result tree fragment as given in Figure 11.2 is obtained. Adding the capital *reference attributes with*

```
C[@capital→City] :-
        result/country→C[@cia:capital→Name and city→City[name/text()=Name]].
```

completes the first step.

XML-QL supports a similar functionality by using *skolem functions* for generating ids (cf. Section 3.5, page 37); here the order of children is given by the global order of all nodes in a document.

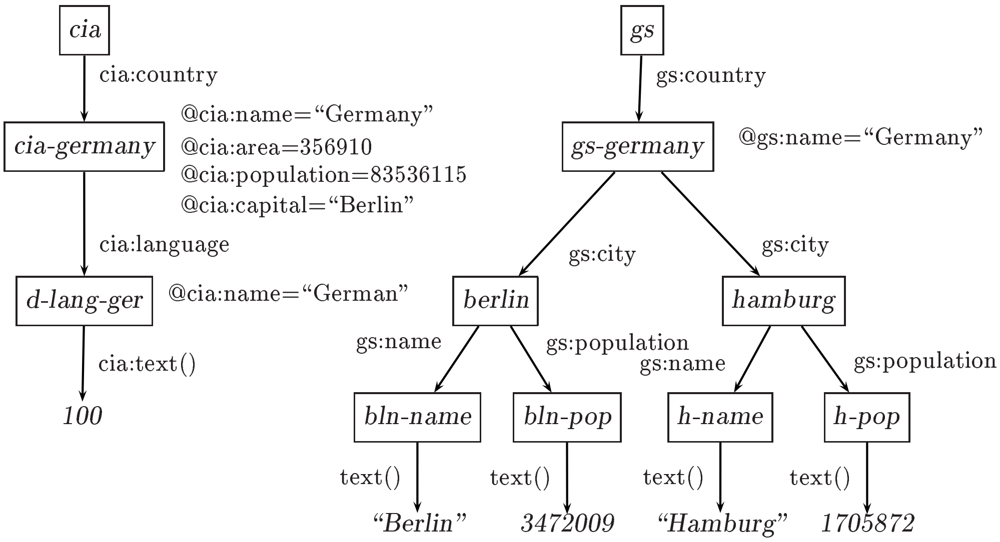

Figure 11.1: Element fusion – before

11.7 Integration Strategies: Summary

The above strategies of *element fusion*, *linking*, and *synonyms* allow for powerful integration concepts for generating a result tree (or even several result trees) from a set of sources (e.g., demonstrated by the case study described in Section 12). When the integration and restructuring process is completed, the projection strategy described in Section 11.1 is used to define a *result view* of the internal database. The result view is again a tree, rooted at a given element.

A crucial feature of a data integration language is that these tasks can be specified in an intuitive, understandable way, and that the language is powerful enough to allow for short and concise statements. The DOM model is not suitable for such operations:

- elements in a tree cannot be linked to another parent - thus, only copying is possible for restructuring, which raises the problems described in Section 8.5.
- every element can only have one unique parent, thus there is no way to define views as "overlays" on the source trees,
- every element has a unique name. Thus, expecially synonyms (which provide a simple, but powerful means) are not supported.

The edge-labeled data model underlying the present approach which also makes names first-class citizens of the model (and the language, respectively), supports data integration by *equating* on different levels:

- by equating *elements*, which represent the same real-world entity in different sources ("*fusing*" objects), these can be made a single element in the internal database, and
- by equating *names*, synonyms for properties can be defined which allow for an efficient integration of properties from several original namespaces.

Note that by using namespaces, the parent *wrt. a given namespace* is in general unique. Thus, namespaces also support the implicit definition of *views*. The namespaces can also be used to filter the properties of the original elements: for a fused element, the expression /child::*namespace*:* selects the children which originate from the source which has been associated with *namespace*.

The use of the integration strategies is illustrated in the MONDIAL case study described in Section 12.

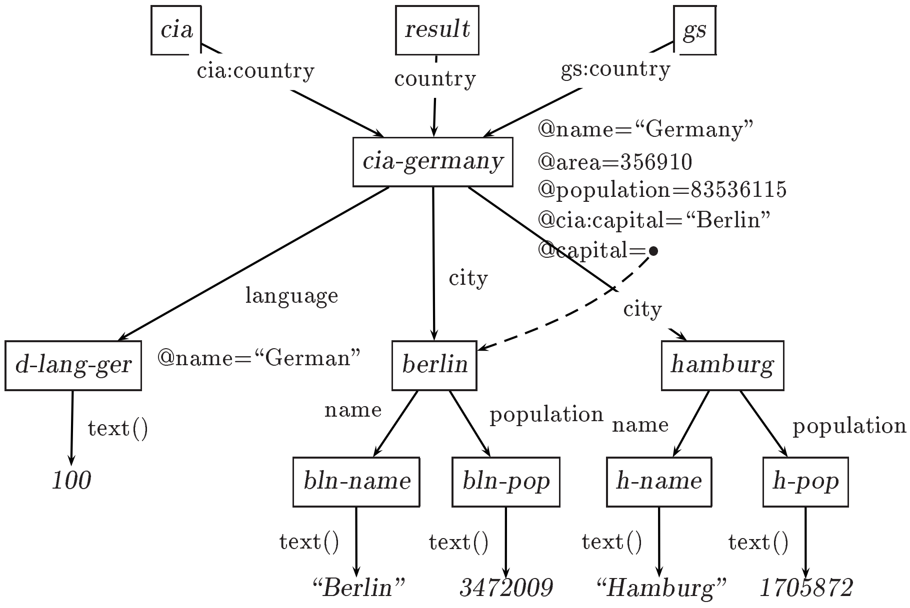

Figure 11.2: Element fusion – after

11.8 Checking Dangling References

As already stated, result trees in most XML querying languages may in general contain dangling references when elements are deep-copied from the source tree(s), and the targets of references are not copied.

The XPathLog result tree defined by the projection strategy using a signature follows only the children relation when collecting tree nodes. Thus, dangling references can occur in the result view. The following XPathLog program checks if the tree rooted in the node *result* and projected wrt. a stored signature contains dangling references:

```
reachable(result).
reachable(Y) :- reachable(X), X/M→Y, X isa C, C[M⇒D], Y isa D.
?- sys.strat.doIt.
dangling :- reachable(X), X isa C, C[@A⇒RC], not C subcl literal,
            X[@A→V], V isa RC, not reachable(V).
```

11.9 Combining Data and Schema Information

For representing metadata, XML provides the concepts of DTDs and XML Schema [XML99c]. DTDs can be transformed into signature atoms by the mapping described in Section 10.2.1. Nevertheless, the DTD data does not actually become part of the XML (graph) database.

In contrast, XML Schema documents are valid XML instances, thus they can be mapped directly to XML trees in the database, and associated with an entry constant. Then, the integration rules can homogeneously use the data trees and the metadata trees.

Example 11.8 (Combining Data and Metadata Trees)
Consider the following excerpt of a XML Schema specification of the cia *source (for the DTD, see Appendix D.1):*

```
<schema xmlns ='http://www.w3.org/1999/XMLSchema' ...>
<complexType name='cia'>
    <element name='continent' type='continent_T' minOccurs='1' maxOccurs='*'/>
    <element name='country' type='country_T' minOccurs='1' maxOccurs='*'/>
</complexType>
<complexType name='continent_T' > ... </complexType>
<complexType name='country_T'>
    <attribute name='name' type='CDATA' use='required'/>
    <attribute name='continent' type='IDREF' use='required'/>
    ⋮
    <element name='borders' minOccurs='0' maxOccurs='*' >
      <complexType base='string' derivedBy='extension'>
        <attribute name='country' type='IDREF' use='required'/>
      </complexType>
    </element>
</complexType>
```

After adding this XML tree to the database under the constant ciaSchema*, it can be mapped to signature atoms as described in Section 10.2.2 and Appendix C. Then, this knowledge can be used for detecting the classes of elements (in this step, it is also possible to validate the instance wrt. the XML Schema):*

```
cia isa cia.
E isa ET :- Parent isa PT, Parent[SE→E], PT[SE⇒ET].
```

Then, e.g., the target types of reference attributes can be detected and added to the XML Schema tree:

```
A[@target→TTS] :-
    ciaSchema//complexType/element[@name→ENS and @type→ETS],
    ciaSchema//complexType[@name→ETS]/attribute→A[@name→ANS and @type="IDREF"],
    % now ANS is an IDREF attribute name of elements with name ENS,
    string2Object(ENS, EN), string2Object(ETS,ET), string2Object(ANS,AN),
    % two-way built-in mapping predicate, e.g., string2Object("country",country) holds
    cia//EN[@AN→Target], Target isa TargetType, string2Object(TTS, TargetType).
```

The above rule makes the following extensions to the internal representation of the ciaSchema *tree:*

```
<complexType name='country_T'>
    <attribute name='continent' type='IDREF' target='continent' />
    ⋮
    <element name='borders' ...>
      <complexType ...>
        <attribute name='country' type='IDREF' target='country' />
      </complexType>
    </element>
</complexType>
```

As the above example shows, identifying *complexType* elements with class names in $\mathcal{N}_{\mathcal{CE}}$ allows for complex interaction between data level and meta level. Similarly, e.g., *links* between elements and schema elements representing element types can be added to the database.

Further reasoning on the metadata level can be performed when ontologies which are accessible in XML format are also added as XML trees to the database in order to guide the integration process. In this setting, rules can be specified which use (i) data, (ii) metadata like XML Schema, and (iii) additional ontology databases. Then, XPathLog rules can be used for reasoning on the meta-level, and then these results can be exploited for integrating the data given by the instances:

- Metadata + ontology: search for related concepts in different data sources, and identify concept overlappings and disjoint parts which extend each other.
- Data + results from above + graph matching algorithms: identify data overlappings (e.g., in a database on cities in european countries, and a database on economics in the G7 countries) and use them for integrating databases, also using analogy reasoning.

12 Mondial: THE CASE STUDY IN INTEGRATION

The practicability of the approach is illustrated by integrating the MONDIAL database from the XML representations of its sources (which have been created by FLORID wrappers in [May99a]); the DTDs can be found in Appendix D, the complete case study with the source files is available at [May01a]. The data sources have already been described in the introduction. Each of the sources is associated with a namespace:

The CIA World Factbook: The CIA World Factbook Country Listing (cia:, `http://www.odci.gov/cia/publications/pubs.html`) provides political, economic, and social and some geographical information about the countries.

A separate part of the CIA World Factbook provides information about political and economical organizations (orgs:).

Here, the data sources overlap by the membership relation: with every organization, the member countries are stored in orgs by name (using the same names as in the cia part).

Global Statistics: Cities and Provinces: The *Global Statistics* data (gs:, `http://www.stats.demon.nl`) provides information about administrative divisions (area and population, sometimes capital) and main cities (population with year, and province).

The information in gs is grouped by countries, using the same country names as in cia. Many cities are capitals of countries or seats of political organizations; here already different names of the cities can be used.

Qiblih: Geographical Coordinates: The Qiblih pages (qiblih:, `http://www.bcca.org/misc/qiblih/latlong.html`) provide the geographical coordinates of many cities around the world.

For many of the cities given in gs, the geographical coordinates can be found here, but there are also cities which are contained in only one of the databases.

The Terra Database: The TERRA[1] database (terra:) also contains information about countries, administrative divisions, cities, and organizations, and additionally geographical data about waters, mountains, deserts etc.

TERRA describes the situation in 1987. Thus, only some information can be used: population of cities in 1987, and the information about geographical objects (in which country and province they are located). The problem here is that some countries changed in the meantime. Additionally, TERRA uses german names for countries.

Thus, another data source relates english and german names and car codes:

Country Names and Codes: An additional Web page (codes:) has been used which gives the country names in different languages and the country codes.

[1]derived from the SQL training TERRA database of the *Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe.*

The Integration Process

The integration program in XPathLog is described below, applying the strategies described in the previous section. The integration process shows that actually the above strategies are used for the "generic" tasks but have to be complemented by application-specific rules for handling details, exceptions, inconsistencies, and sometimes incomplete information.

Preparations.
During the execution, the computation – especially the use of equating objects – is monitored.

```
?- sys.theOMAccess.debugOn.
?- sys.theOM.eqTraceOn. % trace derived equalities
```

First, the data sources are parsed as XML trees with their namespaces and associated to constants:

```
gs[@xml->"file:/home/may/Mondial/Mondial-Sources/gs.xml" isa url].
cia[@xml->"file:/home/may/Mondial/Mondial-Sources/cia.xml" isa url].
orgs[@xml->"file:/home/may/Mondial/Mondial-Sources/orgs.xml" isa url].
terra[@xml->"file:/home/may/Mondial/Mondial-Sources/terra.xml" isa url].
qiblih[@xml->"file:/home/may/Mondial/Mondial-Sources/qiblih.xml" isa url].
codes[@xml->"file:/home/may/Mondial/Mondial-Sources/codes.xml" isa url].
mondial.system = "mondial-3.0.dtd".
mondial[@dtd->"file:/home/may/Mondial/Mondial-Sources/mondial-3.0.dtd" isa url].
?- sys.strat.doIt.
U.parse@(xml,S) :- S[@xml->U].
S = X :-  S[@xml->U], U.parse@(xml,S) = Doc, Doc[S:S->X].
```

Now, the constants gs, cia etc. are root nodes of separate input trees. The result DTD is parsed (generating signature atoms) and a result tree root node is defined:

```
U.parse@(dtd) :- mondial[@dtd->U].
result isa mondial.
?- sys.strat.doIt.
?- sys.garbageCollection.
```

Many properties are taken directly from the sources to the result by defining *synonyms*. Note that the following steps do not effect any tree, but just prepare access methods which become usable in the result view when their host objects are linked.

```
gs:population = population.
gs:country = country.
gs:province = province.
gs:capital = capital.
gs:area = area.
gs:population = population.
gs:year = year.
gs:name = name.
gs:text() = text().
cia:name = name.
cia:borders = border.
cia:ethnicgroups = ethnicgroups.
cia:religions = religions.
cia:languages = languages.
cia:area = area.
cia.population = population.
```

```
cia:datacode =  datacode.
cia:total_area = total_area.
cia:population_growth = population_growth.
cia:infant_mortality= infant_mortality.
cia:gdp_agri = gdp_agri.
cia:gdp_ind = gdp_ind.
cia:gdp_serv = gdp_serv.
cia:gdp_total = gdp_total.
cia:inflation = inflation.
cia:indep_date = indep_date.
cia:government = government.
codes:car_code = car_code.
qiblih:longitude = longitude.
qiblih:latitude = latitude.
orgs:abbrev = abbrev.
orgs:name = name.
orgs:established = established.
?- sys.strat.doIt.
```

From the TERRA source, we cannot reuse complete properties but have to filter wrt. the types they apply to. All geographical entities with their properties are taken from TERRA:

```
terra:mountain = mountain.
  terra:height = height.
  M[@longitude->L1 and @latitude->L2] :-
    terra/mountain->M[@terra:longitude->L1 and @terra:latitude->L2].
terra:desert = desert.
  D[@area->A] :- terra/desert->D[@terra:area->A].
terra:island = island.
  I[@area->A] :- terra/island->I[@terra:area->A].
terra:lake = lake.
  L[@area->A] :- terra/lake->L[@terra:area->A].
terra:river = river.
  terra:to = to.
    terra:type = type.
    terra:water = water.
  terra:length = length.
terra:sea = sea.
  terra:depth = depth.
D[@name->A] :- terra/desert->D[@terra:name->A].
D[@name->A] :- terra/mountain->D[@terra:name->A].
D[@name->A] :- terra/island->D[@terra:name->A].
D[@name->A] :- terra/lake->D[@terra:name->A].
D[@name->A] :- terra/river->D[@terra:name->A].
D[@name->A] :- terra/sea->D[@terra:name->A].
?- sys.strat.doIt.
```

Some cities are not present in any of the sources, but are needed as seats for organizations or as capitals:

```
C/city[name->"Abidjan"] :-
      gs/gs:country->C[gs:name/gs:text()->"Cote dIvoire"].
C/city[name->"Jeddah"] :-
      gs/country->C[gs:name/gs:text()->"Saudi Arabia"].
C/city[name->"El Aaiun"] :-
```

```
      gs/gs:country->C[gs:name/gs:text()->"Western Sahara"].
C[@capital->E] :-
      gs/gs:country->C[gs:name/gs:text()->"Western Sahara"],
      C/city->E[name->"El Aaiun"].
```

Integration of objects. Now, the *construction* of the result tree by linking elements and properties is started.
The continents are taken from gs:

```
result[continent->C] :- gs/gs:continent->C.
?- sys.strat.doIt.
```

Countries. The CIA country listing provides the base for including countries into the result tree. Only those countries are relevant, where a capital is given in cia (excluding many irrelevant, sometimes even unpopulated territories):

```
C[@cia:rel_country->1] :- cia/cia:country->C[@cia:capital].
?- sys.strat.doIt.
```

Fusing CIA and GlobalStatistics Countries. The GlobalStatistics source uses the same country names as CIA. The corresponding country elements are fused and incorporated into the result tree, collecting the properties of both original elements:

```
result[country->C1], C1 = C2 :-
   cia/cia:country->C1[@cia:rel_country and @cia:name->N],
   gs/gs:country->C2[gs:name/gs:text()->N].
C[@name->N] :- result/country->C/@cia:name->N.
result[country->C[@cia:rel_country->1 and @name->"Serbia and Montenegro"]] :-
   gs/gs:country->C[gs:name/gs:text()->"Serbia and Montenegro"].
?- sys.strat.doIt.
```

Cities. The GlobalStatistics source contains detailed information about cities. All cities from gs are taken as result cities. Some cities are stored with two different names (once as a city, and once mentioned as the capital of a province or country): names like "Mexico", "Mexico City" and "Ciudad de Mexico" denote the same object – same for Panama:

```
result[city->C[@name->N]] :- gs/gs:city->C[name/text()->N].
?- sys.strat.doIt.
C1 = C2 :-
   gs/gs:city->C1[@gs:country->C and @name->N1],
   gs/gs:city->C2[@gs:country->C and @name->N2],
   strcat(N1," City", N2).
C1 = C2 :-
   gs/gs:city->C1[@gs:country->C and @name->N1],
   gs/gs:city->C2[@gs:country->C and @name->N2],
   strcat("Ciudad de ", N1, N2).
?- sys.strat.doIt.
```

Fusing CIA and GlobalStatistics Cities. The *capital* property of *countries* is made a reference to the *city* objects: in the CIA country listing, the only cities mentioned are the capitals. Some of them are stored under different names than in GlobalStatistics.

```
city_synonym("Bucharest","Bucuresti").
city_synonym("Warsaw","Warszawa").
city_synonym("New Delhi","Delhi").
city_synonym("Tashkent","Toshkent").
```

```
city_synonym("Addis Ababa","Addis Abeba").
city_synonym("La Hague","s'-Gravenhage").
C1 = C2 :-
   gs/gs:city->C1[@gs:country->Ctry and @name->N1],
   gs/gs:city->C2[@gs:country->Ctry and @name->N2],
   city_synonym(N1,N2).
?- sys.strat.doIt.
    not C/@gs:main_cities.
?- sys.strat.doIt.
```

Provinces. The provinces from GlobalStatistics are added to the result tree if they belong to relevant countries. Recall that the *country* attribute of GlobalStatistics provinces already points to the integrated (fused) country objects.

```
result[province->P] :- gs/gs:province->P/@country[@cia:rel_country].
?- sys.strat.doIt.
```

Now, the integration of the CIA country listing and the GlobalStatistics source is already finished, mainly using *object fusion* and linking elements to the result tree. The "detailed" properties of the objects have been handled above by just defining synonyms used in the result tree (via the result signature).

Organizations and Memberships. All organizations from the CIA organizations listing are linked into the result tree:

```
result[organization->O] :- orgs/orgs:organization->O.
```

Note that the CIA country data and the CIA organizations data is not fully consistent: there are 71 memberships which are not mentioned in the country data, and there are 6 organizations which are mentioned in the country data which do not exist in the organization data.

Since the organizations data is more detailed (membership type), the membership information is taken from there. The membership lists contain the country names. Based on this information, reference attributes are defined. Note that there are members which are no official countries, mostly international commissions or local interest groups. In case the member list enumerates a country both as a member and as an associate member/observer (IFRCS), only the full membership is taken:

```
O/members[@type->"member" and @country->C] :-
   result/organization->O/orgs:member_names[@orgs:type="member"]/orgs:text()->CN,
   result/country->C[@name->CN].
?- sys.strat.doIt.
O/members[@type->T and @country->C] :-
   result/organization->O/orgs:member_names[@orgs:type->T]/orgs:text()->CN,
   not O/orgs:member_names[@orgs:type="member"]/orgs:text()->CN,
   result/country->C[@name->CN].
```

The seats of organizations (mentioned by country and name in the CIA organizations listing) are linked to the respective cities:

```
O[@seat->Cty] :-
   result/organization->O[@orgs:seatcity->N and @orgs:seatcountry->CN],
   result/city->Cty[@name->N and @country/@name->CN].
?- sys.strat.doIt.
```

Fusing CIA and codes source. In the next step, the codes data source is integrated which contains the english, german, and local names of the countries and their car-codes. For most countries, the CIA and english name coincides:

```
C1 = C2 :-
   codes/codes:country->C1/codes:name[@codes:language = "english"]/codes:text()->N,
   result/country->C2[@name->N].
?- sys.strat.doIt.
```

For other countries, the CIA name coincides with the german or local name:

```
C1 = C2 :-
   codes/codes:country->C1/codes:name[@codes:language = "german"]/codes:text()->N,
   result/country->C2[@name->N].
C1 = C2 :-
   codes/codes:country->C1/codes:name[@codes:language = "local"]/codes:text()->N,
   result/country->C2[@name->N].
```

As a side effect, for all result countries, also the attribute country/@car_code is defined (taken from codes).

Linking Qiblih-Cities to Countries. In the next steps, the qiblih source is integrated which contains geographical coordinates for many cities. First, the qiblih cities are linked to the appropriate country and province elements.

In qiblih, the countries are only mentioned by a string-valued attribute city[@qiblih:country⇒country-name]. All qiblih cities for which this attribute matches a name which occurs as country/@name are linked to the respective country.

```
Cty[@country->C] :-
   qiblih/qiblih:city->Cty[@qiblih:country->N], result/country->C[@name->N].
```

There are some CIA capitals which are also contained in qiblih, but not with the exact country name. They are matched by city name, excluding the US and Canadian cities (which can be detected since for them, city/@qiblih:province is defined).

```
Cty[@country->Country] :-
   qiblih/qiblih:city->Cty[@qiblih:name->N], not Cty/@country,
   not Cty/@qiblih:province,
   result/country->Country[@cia:rel_country]/@gs:capital[@name->N].
```

For US and Canadian cities, qiblih knows the provinces (by name). These cities are also linked to the corresponding province object using the province name:

```
Cty[@province->P] :-
   qiblih/qiblih:city->Cty[@country->S and @qiblih:province->N],
   gs//gs:province->P[@gs:country->S and @gs:name->N].
```

Merging Qiblih-Cities with GlobalStatistics main cities. Note that the qiblih cities are not yet integrated to the result tree. Cities which exist in qiblih and in CIA/GlobalStatistics are now merged. Qiblih also contains information about cities which are not mentioned in the CIA/GlobalStatistics sources; these cities are ignored.

If no province is known in qiblih (all except US and CDN), cities are matched by *name* and *country*:

```
C1 = C2 :- qiblih/qiblih:city->C1[@qiblih:name->N and @country->Ctry
                                  and not @qiblih:province],
           result/city->C2[@name->N and @country->Ctry].
?- sys.strat.doIt.
```

In the USA, qiblih and GlobalStatistics main cities are matched by (*name*,*state*,*country*):

```
C1 = C2 :- qiblih/qiblih:city->C1[@qiblih:name->N and not @name and
                                 @country->Ctry and @province->P],
           result/city->C2[@name->N and @province->P and @country->Ctry].
?- sys.strat.doIt.
```

For Canada, qiblih provides provinces, but GlobalStatistics does not. Cities are matched by *name* and *country*:

```
C1 = C2 :- qiblih/qiblih:city->C1[@qiblih:name->N and not @name and
                                 @country->CDN[@name="Canada"]],
           result/city->C2[@name->N and @country->CDN].
?- sys.strat.doIt.
```

The fused cities now have a *population* property (with *year*) from GlobalStatistics, and (*longitude*,*latitude*) data from qiblih.

Terra. The integration of the TERRA database is a more complex task: Its countries, provinces, and cities overlap with the CIA and GlobalStatistic ones. TERRA represents the state of the world in 1987, thus some of its information is outdated. Some of its cities and provinces (which are still existing) which are not present in the other databases are added. The population information of 1987 is added for all cities (annotated with the @year="1987" attribute). Additionally, TERRA contains information about geographic objects like mountains and waters which is completely added to the result. For these objects, it is stored to which countries and provinces they belong, thus, a mapping from the 1987 countries to the 1997 countries must be implemented, based on knowledge about the application domain. Since TERRA uses the german names of countries, the additional information of the codes source proves useful here.

Terra Countries. The TERRA countries are fused with the CIA/result countries. In most cases, this can be done using the car_code attribute.

```
L = C :- terra/terra:country->L[@terra:code->Code],
         result/country->C[@car_code->Code].
?- sys.strat.doIt.
```

Others are fused by matching the TERRA name with the german name given in the codes source, or with the name given by CIA:

```
L = C :- terra/terra:country->L[@terra:name->N and not @car_code],
         result/country->C/codes:name[@codes:language ="german"]/codes:text()->N.
L = C :- terra/terra:country->L[@terra:name->N and not @car_code],
         result/country->C/@name->N.
```

If the name of a capital is the same in CIA/GlobalStatistics and TERRA, the countries are also the same:

```
L = C :- terra/terra:country->L[@terra:capital->CN and not @car_code],
         result/country->C/@capital/@name->CN.
?- sys.strat.doIt.
```

The only remaining TERRA countries are those of the USSR, GDR and CS (countries which have been dissolved).

Fusing Administrative Divisions. TERRA contains information about cities belonging to administrative divisions which are not contained in GlobalStatistics. This information is also integrated. Some of the administrative divisions can be fused by country/name:

```
L = P :- terra/terra:province->L[@terra:country->CC and @terra:abbrev->AB
                                and @terra:name->N],
         not CC = AB,
         result/province->P[@gs:name -> N]/@country[@car_code->CC].
?- sys.strat.doIt.
```

For the remaining administrative divisions, the capital coincides with the capital of an administrative division in GlobalStatistics of the same country (which is already merged). Since TERRA contains the country itself as a province if no other provinces are known, these have to be excluded:

```
L = P :- terra/terra:province->L[@terra:country->CC and @terra:capital->CN],
         not result/province->L,
         count{X [CC]; terra/terra:province->X[@terra:country->CC]} > 1,
         result/province->P[@capital[@name->CN] and
                           @country/@car_code->CC].
?- sys.strat.doIt.
```

The provinces of Finland and Norway are only contained in TERRA. They are made administrative divisions of their countries, the capital is made a city, and its reference attributes to the province and to the country are set:

```
p(finland,"SF"), finland = C :- result/country->C[@car_code->"SF"].
p(norway,"N"), norway = C :- result/country->C[@car_code->"N"].
?- sys.strat.doIt.
result[city->Cap[@name->CN and @country->Ctry and @province->L]],
result[province->L[@country->Ctry and @name->N and
                  @capital->Cap and @population->E]] :-
   p(Ctry,CC),
   terra/terra:province->L[@terra:country->CC and @terra:abbrev->AB and
           @terra:name->N and @terra:capital->CN and @terra:pop->E],
   terra/terra:city->Cap[@terra:country->CC and
           terra:province/terra:text()->AB and @terra:name->CN].
```

For the former USSR, all its administrative divisions (the SSRs) are now independent and are fused with the respective *country* objects from CIA and GlobalStatistics. Here, the matching is done by the first three letters of the (german) name. Note that this leads to objects which are countries and terra:provinces.

```
L = C :- terra/terra:province->L[@terra:name->N1 and @terra:country->"SU"],
         result/country->C/codes:name[@codes:language="german"]->N2,
         not terra/terra:country->C,
         pmatch(N1,"/\A(...)/","$1",N1short),
         pmatch(N2,"/\A(...)/","$1",N2short), N1short = N2short.
?- sys.strat.doIt.
```

Continent Information. TERRA contains information about the distribution of countries over continents in its *encompassed* property. Note that this is not contained in CIA or GlobalStatistics for countries which belong to two continents (Russia, Egypt, Turkey).

```
C/encompassed[@continent->CT and @percentage->Perc] :-
   result/country->C/terra:encompassed[@terra:continent->CN]/terra:text()->Perc,
   result/continent->CT[@gs:name->CN].
```

```
C/encompassed[@continent->Asia and @percentage->80],
C/encompassed[@continent->Europe and @percentage->20] :-
   result/country->C[@name->"Russia"],
   result/continent->Europe[@gs:name->"Europe"],
   result/continent->Asia[@gs:name->"Asia"].
?- sys.strat.doIt.
```

All other countries belong with 100% to the continent which is given in CIA/GlobalStatistics:

```
C/encompassed[@continent->Cont and @percentage->100] :-
   result/country->C[@gs:continent->Cont and not encompassed].
?- sys.strat.doIt.
```

Fusing Terra Cities. TERRA contains geographical coordinates for all its cities, many of them are also contained in GlobalStatistics without coordinates and qiblih also does not provide their coordinates. For countries where no administrative divisions are distinguished in TERRA (i.e. terra:city[@terra:country = @terra:province]), cities are merged by *country* and *name*:

```
S = C :- terra/terra:city->S[@terra:name->N and @terra:country->CC and
                             terra:province/terra:text()->CC],
         result/country->Country[@terra:code->CC],
         result/city->C[@country->Country and @name->N].
```

Cities where GlobalStatistics knows the province (US cities and capitals of provinces) are fused using (*name*,*province*,*country*):

```
S = C :- terra/terra:city->S[@terra:name->N and @terra:country->CC and
                             terra:province/terra:text()->AB],
         not CC = AB,
         result/country->Country[@terra:code->CC],
         terra/terra:province[@terra:name->PName and @terra:country->CC
                              and @terra:abbrev->AB],
         result/city->C[@country->Country and @name->N],
         result/province[@gs:name->PName].
?- sys.strat.doIt.
```

There are cities where TERRA knows the province but GlobalStatistics does not know it although the province as an object is known. These can be fused and linked using the TERRA information:

```
S = C, C[@province -> P] :-
   terra/terra:city->S[@terra:name->N and @terra:country->CC and
                       terra:province/terra:text()->AB],
   not CC = AB,
   result/country->Country[@terra:code->CC],
   terra/terra:province->P[@terra:country->CC and @terra:abbrev->AB],
   result/city->C[@country->Country and @name->N and not @province].
?- sys.strat.doIt.
```

TERRA cities which are capitals are fused with the CIA/GlobalStatistics capital of the same country. Here, all countries (i.e., Germany) where the capital changed must be *explicitly* excluded!

```
S = C :- terra/terra:city->S[@terra:name->N and @terra:country->CC],
         result/country[@terra:code->CC and @terra:capital->N and @capital->C],
         not result/city->S, not CC="D".
```

Provinces are handled in the same way:

```
S = C :- terra/terra:city->S[@terra:name->N and @terra:country->CC and
                              terra:province/terra:text()->AB],
         terra/terra:province[@terra:country->CC and @terra:abbrev->AB and
                               @terra:capital->N and @capital->C],
         not result/city->S.
```

Again, the former USSR needs special treatment: capitals of SSRs are now the capitals of the new countries:

```
S = C :- terra/terra:city->S[@terra:name->N and @terra:country->"SU" and
                              terra:province/terra:text()->AB],
         terra/terra:province[@terra:country->"SU" and @terra:abbrev->AB and
                               @terra:capital->N and @capital->C].
?- sys.strat.doIt.
```

Non-capital USSR cities can only be matched by name (note that the former TERRA provinces have been equated with the new countries):

```
S = C :- terra/terra:city->S[@terra:name->N and @terra:country->"SU" and
                              terra:province/terra:text()->AB],
         result/city->C[@country->Country[@terra:country->"SU" and
                                            @terra:abbrev->AB] and
                        @name->N].
?- sys.strat.doIt.
```

Additional Terra Information. TERRA contains some more data which has not yet been integrated, e.g. cities which are not mentioned in any of the other databases, or additional information about cities which are already known.

There are some countries where GlobalStatistics gives no main cities (except the capital), but TERRA gives some additional cities. Then, these cities can be added to the database without any risk to have a city under two different names. This also holds when all GlobalStatistics cities are capitals of provinces and TERRA also knows these provinces/capitals (then the capitals are already fused).
First, all GlobalStatistics cities are determined which are no capitals:

```
C[@notcapital->true] :-
   result/city->C[@country->X], not X[@capital->C],
   not X/gs:adm_divs[@capital->C].
?- sys.strat.doIt.
```

For countries which have provinces in TERRA, and GlobalStatistics knows no non-capital cities (effectively, this implies that GlobalStatistics knows no provinces, these are only N and SF), all TERRA cities can be added with their province information:

```
result[city->S[@country->C and @province->P]] :-
   result/country->C[@terra:code->CC],
   not C/gs:main_cities[@notcapital],
   terra/terra:city->S[@terra:country->CC and terra:province/terra:text()->AB],
   not CC = AB, not result/city->S,
   result/province->P[@country->C and @terra:abbrev->AB].
?- sys.strat.doIt.
```

TERRA cities which are known with their province, where GlobalStatistics knows no non-capital cities in this province, can be added with their province information:

```
result[city->S[@country->C and @province->P]] :-
   result/country->C[@terra:code->CC],
   terra/terra:city->S[@terra:country->CC and terra:province/terra:text()->AB],
   not CC = AB, not result/city->S,
   result/province->P[@country->C and @terra:abbrev->AB],
   N = count{City [P]; result/city->City[@province->P]}, N = 1.
```

For countries (except the former USSR) where GlobalStatistics knows no main cities and TERRA knows no provinces, these cities are added:

```
result[city->S[@country->C]],
C[@gs:main_cities->S] :-          %% other countries
   result/country->C[@terra:code->CC], not CC = "SU",
   N = count{City [C]; C/@gs:main_cities->City}, N = 1,
   terra/terra:city->S[@terra:country->CC and terra:province/terra:text()->CC],
   not result/city->S.
?- sys.strat.doIt.
```

The *longitude* and *latitude* data of TERRA cities is added if there is no other (qiblih) information:

```
C[longitude->Long and latitude->Lat] :-
   result/city->C[@terra:longitude->Long and @terra:latitude->Lat],
   not C/@latitude, not C/@longitude.
?- sys.strat.doIt.
```

Take the city name from TERRA if there is no other name (cities which are taken from TERRA and not equated with any GlobalStatistics city):

```
C[@name->N and name->N] :-  result/city->C[@terra:name->N and not name].
?- sys.strat.doIt.
```

The *population* data of TERRA cities is added if there is no other (GlobalStatistics) information:

```
C/population[@year->87 and text()->E] :-
    result/city->C[@terra:population->E and not population].
?- sys.strat.doIt.
```

For countries where the car-code is not yet given, it is taken from TERRA (currently this is only Burma and French Guiana):

```
C[@car_code->CC] :- result/country->C[@terra:code->CC and not @car_code].
```

Terra geo objects. The geographic objects (rivers, lakes, seas, deserts, mountains, islands) are not described in the other sources. They are added to the result.

```
mountain subcl geo_obj.
desert subcl geo_obj.
island subcl geo_obj.
water subcl geo_obj.
lake subcl water.
river subcl water.
sea subcl water.
?- sys.strat.doIt.
result[Class->Obj] :-  terra/Class->Obj, Class subcl geo_obj.
?- sys.strat.doIt.
```

Terra geo objects locations. For the geographic objects, the country and province information has to be integrated. Since the TERRA countries and provinces are already fused with the result countries and provinces (i.e., the complex identfication task has already been done), this task consists mainly of generating appropriate reference attributes:
First, the countries are determined; for the former USSR, (SU,prov) is changed into (prov,*null*):

```
Obj[@country->C] :-
   terra/Class->Obj/terra:located[@terra:country_code->CC],
   Class subcl geo_obj,
   not CC =  "SU",
   result/country->C[@terra:code->CC].
Obj[@country->C] :-
   terra/Class->Obj/terra:located[@terra:country_code="SU" and @terra:province_id->AB],
   result/country->C[@terra:abbrev->AB].
Obj[country->C] :-
   terra/Class->Obj/terra:located[@terra:country_code="CS"],
   result/country->C[@car_code="CZ"].
?- sys.strat.doIt.
```

As far as provinces are given, this information is also added:

```
L[@country->C and @province->P] :-
   terra/Class/terra:located->L[@terra:country_code->CC and @terra:province_id->AB],
   result/province->P[@terra:abbrev->AB and @country->C/@terra:code->CC].
?- sys.strat.doIt.
```

Additional Data. There are some cities and geo objects whose provinces are not given yet. They are contained in an additional file (created manually): The predicate addprov(*name, pop, code, provincename*) gives the province for the city with name *name* in the country with the car code *code*.

```
?- sys.load@("../Mondial-programs/mondial-addprovs.flp").
format.name="$out[0]=$in[0];
             $out[0]=~tr//aaaaaEeeiiIOooouucn/".
?-sys.strat.doIt.
addprov2(N2,CC,PN2) :- addprov(N,_,CC,PN),
  perl(format.name, N, N2),  perl(format.name, PN, PN2).
?-sys.strat.doIt.
C[@province->P] :-
  addprov2(N,CC,PN),
  result/city->C[@name->N]/@country[@car_code->CC],
  result/province->P[@name->PN]/@country[@car_code->CC].
?-sys.strat.doIt.
Obj[@country->C] :-
  addgeocountry(Geo,Name,Code), Class subcl geo_obj,
  terra/Class->Obj[@name->Name], result/country->C[@car_code->Code].
?- sys.strat.doIt.
```

For province information about geo objects, for every country, a semicolon-list of provinces is given by addgeoprov(*geotype, geoname, countrycode,* $prov_1$;$prov_2$;...).

```
addgeoprov(Geo, Name, Code, ProvName) :-
  addgeo(Geo, Name, Code, Strg),
  pmatch(Strg, "/([A-Za-z][^;]*)/g", "$1", ProvName).
?- sys.strat.doIt.
```

```
Obj/located[@country->C and @province->P] :-
  addgeoprov(Geo, Name, Code, ProvName),
  terra/Class->Obj[@name->Name and @country->C[@car_code->Code]],
  result/province->P[@country->C and @name->ProvName].
?- sys.strat.doIt.
```

Result restructuring. Some properties in the result are restructured internally on XML level:

```
E[text()->N and @percentage->P] :-
   result/country->C/ethnicgroups->E[@cia:name->N and cia:text()->P].
R[text()->N and @percentage->P] :-
   result/country->C/religions->R[@cia:name->N and cia:text()->P].
L[text()->N and @percentage->P] :-
   result/country->C/languages->L[@cia:name->N and cia:text()->P].
?- sys.strat.doIt.
B[@country-> C and @length->L] :-
    result/country/border->B[@cia:country-> C and cia:text()->L].
?- sys.strat.doIt.
```

The *city* and *province* elements which are still direct children of the result root node are linked as a hierarchical structure country/province/city or country/city:

```
C[province->P] :- result/province->P[@country->C].
C[city->City] :- result/city->City[@country->C and not @province].
P[city->City] :- result/city->City[@province->P].
?- sys.strat.doIt.
```

Recall that above, the Mondial DTD has been parsed, yielding an *export signature*. The result tree is now exported according to this signature:

```
?- sys.theOMAccess.export@("xml","mondial-3.0.xml","mondial","result").
```

Lessons Learnt

First, the principal objects (countries, provinces, cities, and organizations) have been collected. The overlapping parts of the corresponding sources (CIA countries, CIA organizations, and GlobalStatistics) were well-defined. Here, the main tasks consisted of

- identifying corresponding elements in overlapping sources, and fusing them, and
- generating references between elements from the non-overlapping parts.

Exploiting the features of XPathLog for

- object fusion,
- multiple parents, and
- synonyms,

the rules could be kept short and declarative.
The integration of the small source providing country codes and names has also been done by fusing objects.

With adding the Qiblih source as complementing knowledge, the "usual" integration problems came up: For all "classes" of real-world objects described in Qiblih, some of the instances were already present in the result tree, some others were not. Additionally, different names were used. Thus, the principal tasks – fusing and linking – were the same, but exceptions had to be dealt with.

The integration of the TERRA source mirrors all problems of "autonomous" data sources: The fact that TERRA represents the world of 1987 can be regarded as an "inconsistency": some of its information must not be used since it conflicts with the "valid" information extracted from the CIA database (e.g., there is no country called "Soviet Union"). Additionally the use of different names for the same objects (german vs. english) is a typical problem which must be solved when integrating arbitrary sources. Similar to the Qiblih problems, the overlappings were not easy to identify: *some* TERRA cities were also contained in the GlobalStatistics/CIA sources; here the 1987 population, and sometimes the geographical coordinates had to be added (if Qiblih did not do it). The same holds for provinces. So, a complete integration required a thorough control which objects were handled in a certain step, and to check why some objects remained still un-integrated (cf. capitals of provinces in Norway and Finland).

Furthermore, knowledge about the application domain was necessary to handle the transition of provinces of the former USSR to independent countries.

Concerning the relationship of the approach with automatical approaches to data integration, the result is that the automatical generation of a program skeleton based on the source descriptions (e.g., as a DTD, in XML Schema, or by RDF) seems to be promising: The integration of CIA countries, CIA organizations, and GlobalStatistics could be done automatically if a suitable ontology is provided. From the knowledge that `country/@capital` and `organization/@seat` must reference cities, and information about potential key values, the rules could be created automatically.

In contrast, the integration of the other sources can probably not be performed automatically in a satisfying way since it requires knowledge not only on the schema, but also on the data and its real-world background.

Thus, data integration (not only in XML) requires a powerful language and a flexible data model. In XPathLog,

- the operations "fusion" and "linking", and synonyms,
- navigation-based (in this case, XPath-based) expressions which can bind variables to arbitrary nodes which are traversed by the navigation expression,
- the graph-based data model, and
- the projection mechanism

are the components which make up such a language. The nature of an XPathLog program as a list of rules allows for grouping rules which together handle a certain task. The programs are modular which also allows for adapting them to potential changes in the source structure.

13 HANDLING OF XLINKS

The XPointer and XLink working drafts [XPt00, XLi00] (see also Section 3.8) specify how to *express* inter-document links in XML. There is not yet an official proposal how to *handle* XLinks in queries and applications.

As a running example in this section, again the distributed version of the MONDIAL database given in Section 3.8 is used. The section starts with general considerations on XLinks, and then describes an approach to handle XLinks in XPathLog.

Example 13.1 (Querying XLinks)
Consider again the query given in Example 3.4:

> *"Select all names of cities which are seats of an organization and the capital of one of its members."*

For the distributed database, the query has to be formulated in a different way. The entry point for the query in the distributed database is the document mondial-memberships.xml*, iterating over all* membership *elements (e.g., ("D", "EU")),*

- *following the link to the corresponding organization in* mondial-organizations.xml*, searching for the* <organization id="org-EU"> *element, which contains a (simple) xlink subelement* seat*. The target of the link is a city element in* mondial-cities-B.xml*; the text contents of its name has to be retrieved.*
- *following the link to the corresponding country in* mondial-countries.xml*, searching for the* <country car_code="D"> *element, which contains another (simple) xlink subelement* capital*. The target of the link is the city element*

  <city > <name>Berlin</name> </city>

  in mondial-cities-D.xml*; again, the text contents of its name has to be retrieved and compared with the result from the first search.*

13.1 General Considerations on Traversing XLinks

13.1.1 Data Model

Similar to `IDREF(S)`, there are several strategies how a query language may support the resolving of XLinks. For illustration, the XQuery syntax is used.

Uninterpreted. Current XML querying tools are restricted to evaluate XPath expressions wrt. the DOM-resident XML tree (which may contain several documents). When a query is parsed, the documents occurring in expressions of the form document(*url*) are added to the DOM to be evaluated.

Elements with XLink functionality are handled like "normal" elements. They simple have an attribute xlink:href which has a value which accidentally contains an url part (before the "#") and an XPath/XPointer part (after the "#"). Resolving of the link must be done manually: the XPointers giving the urls and XPath expressions are decomposed at runtime and are used to formulate the subsequent subquery.

Remark 13.1
Neither the XQuery working draft, nor the Quilt documentation specifies if statements of the form

FOR $url IN .../*linkelement*/@href,
    $var IN document($\boxed{\text{\$url}}$)/...

are allowed. For this section, we deliberately assume that it is allowed to use variables at document position in subsequent FOR – LET clauses, in fact constructing a query in Dynamic SQL or JDBC style:

```
FOR $xpointer IN ...[xlink:type = "simple"]/@xlink:href,
LET $url = string-before("#",$xpointer),
      $path = string-after("#",$xpointer),
FOR $result in document($url)/$path
      ⋮
```

Such queries are not supported by the current querying languages, and they also are not really declarative. The above behavior can be implemented by XSLT [XSL99] patterns which resolve an xlink:href into the url and the XPointer, and then use the document() extension function. □

Explicit Dereferencing. Similar to IDREF(S) attributes which carry an *internal* semantics allowing to dereference them, XLink elements can be equipped with a similar internal semantics. Quilt/XQuery [XQu01] define the IDREF(S) dereferencing operator "→ *nametest*" as described in Example 3.4. Similar, a dereferencing operator "⇝ *nametest*" for following XPointers in xlink:href attributes can be defined for *inter-document navigation*.

Example 13.2 (Querying XLinks: Explicit Dereferencing)
Using explicit dereferencing, the above query can be formulated as follows. For illustrating in which XML document the current navigation step is applied, the namespaces ms: *(in* mondial-memberships.xml*),* org: *(in* mondial-organizations.xml*),* ctry: *(in* mondial-countries.xml*), and* cty: *(in* mondial-city-code.xml*) are used:*

```
FOR $ms IN document("mondial-memberships.xml")//ms:membership,
      $org IN $ms/@xlink:to→ms:organization/@xlink:href⇝org:organization,
      $abbrev IN $org/@org:abbrev,
      $seatname IN $org/org:seat/@xlink:href⇝cty:city/cty:name/text(),
      $capname IN $ms/@xlink:from→ms:country/@xlink:href⇝ctry:country/
                        ctry:capital/@xlink:href⇝cty:city/cty:name/text(),
WHERE $capname = $seatname
RETURN <result org = $abbrev city = $seatname/>
```

The details of the implementation of the XLink dereferencing operator are the subject of the following sections.

Transparent XLinks. In the above example, an explicit operator for dereferencing XLinks has been introduced. Another possibility is to regard link elements to be *transparent*, i.e., the (abstract) data model should silently replace XLink elements of the types xlink:simple, xlink:locator, and xlink:arc by the result sets of their XPointers (for xlink:type=arc elements, the *from* and *to* attributes are replaced by reference attributes whose name is the *xlink:role* type of the corresponding *xlink:locator* nodes (if no role is given, the name of the locator node is taken instead)). Note that these partially "virtual" elements must accumulate the attributes and contents of the original XLink element (in case it has non-xlink contents) and of the linked element.

Example 13.3 (Querying Transparent XLinks)
With transparent XLinks, the virtually linked XML trees look as follows:

In mondial-memberships.xml*, the* <country xlink:type="locator" ...> *elements are "instantiated" with the* document("mondial-countries.xml")/country *elements resulting from the* xlink:href *attribute; analogously for the* <organization xlink:type="locator" ...> *elements. In the* <membership xlink:type="arc"> *elements, the* xlink:from *attribute is replaced by a* country *reference attribute (taking the name of the element referenced by the* xlink:from *attribute); analogously for the* xlink:to *attribute:*

```
<memberships>
    <!-- xlink:type="locator" xlink:href="document('.../countries.xml')/id('D')" -->
    <country id="D">
        <!-- with attributes and contents of document(".../countries.xml")/id('D') -->
    </country>
    <!-- xlink:type="locator" xlink:href="document('.../organizations.xml')/id('org-EU')" -->
    <organization id="org-EU">
        <!-- with attributes and contents of document(".../organizations.xml")/id('org-EU') -->
    </organization>
    <membership
        <!-- xlink:from="D" xlink:to="org-EU" + types of the from/to elements -->
        country="D" organization="org-EU" membership_type="member"/>
    ⋮
</memberships>
```

Similarly, in mondial-organizations.xml*, the* <seat xlink:type="simple"> *subelements are resolved by the corresponding* document("mondial-city-code.xml")/country *elements:*

```
<organization id="org-UN">
  <seat
    <!-- href="file:cities-USA.xml#//city[name/text()='New York']"-->
    <!-- all attributes of
        document(".../mondial-cities-USA.xml")//city[name/text()='New York'] --> >
    <!-- element contents of
        document(".../mondial-cities-USA.xml")//city[name/text()='New York'] -->
  </seat>
  ⋮
</organization>
```

Analogously, the capital *subelements in* mondial-countries.xml *are (virtually) resolved.*
Then, the query in the virtual linked trees reads as

```
FOR $ms IN document("mondial-memberships.xml")//ms:membership,
    <!-- $ms is of the form   <membership country="D" organization="org-EU"
                                  membership_type="member"/> -->
    $org IN $ms/@ms:organization,                <!-- dereferencing the arc element -->
    $abbrev IN $org/@org:abbrev,
    $seatname IN $org/org:seat/cty:name/text(),    <!-- dereferencing the simple link -->
    $capname IN $ms/@ms:country/ctry:capital/cty:name/text()
                                  <!-- dereferencing the arc and the simple link -->
WHERE $capname = $seatname
RETURN <result org = $abbrev city = $seatname/>
```

Using the transparent model with virtually linked trees in XPathLog (where the linking may be non-virtually), (the constant memberships *represents the root of the* mondial-memberships.xml *tree), the query reads as*

?- *memberships*//ms:membership
[@ms:organization[@org:abbrev→A and org:seat/cty:name/text()→N] and
@ms:country/ctry:capital/cty:name/text()→N].

Another question is, whether virtual subelements via XLinks should be considered as descendants. For transparency reasons, it is sometimes desirable to regard XLink-ed XML instances as one. Then, queries are robust against suitable restructuring of information servers. On the other hand, the set of descendants may become arbitrary large. Additionally, navigation using "//" has to resolve (transitively) all XLinks. Thus, it is desirable that the behavior of an XLink in this aspect can be defined.

So far, the considerations effect only the abstract data model and the syntax of a querying construct.

13.1.2 XML Information Server Cooperation

An XPointer does in general not address a complete document, but gives an XPath expression[1] which has to be evaluated to yield a node set. When such XLinks are resolved in distributed documents provided by autonomous servers all over the Web, one of the main issues is, where the computations are done:

- If only a plain XML document (e.g., as ASCII) is provided, the client which initiated the query has to access the document and evaluate the XPointer expression by itself.
- If an information provider does not publish the XML document, but allows access only by XPath expressions against his database, the answer must be computed by the server.

In general, when an XLink is found, it is not known to the processing instance which of the above cases applies. As a straightforward strategy, the client will first try to send the complete XPointer via HTTP to the server where it will be answered if the server is an XML database system. If the query is not answered, the client will try to access the XML document, parse it into its own database, evaluate the XPointer and process the result set. More involved strategies which are tailored to the needs of a given application are described in the following.

13.1.3 Proposal: Evaluation Strategies for XLinks

The XLink working draft [XLi00] defines several attributes for XLink elements which specify the *behavior* of the XLink element, i.e., *when* it should become "activated" and *what* happens then. In the current version, this behavior is tailored to the use of XLinks when browsing, it does not cover the requirements of querying XML instances. Given an XLink element of the form

<foo xlink:type="simple" href="*url#xpointer*" .../>

there are several strategies what to do from the querying and database aspect:

Activating Event: The activating event of XLinks is considered in the XLink working draft with the xlink:actuate attribute: the value "auto" states that the link is activated when it is parsed, whereas "user" states that it is activated by the user (HTML: clicking).

In our context, "auto" would mean that the XPointer is evaluated when the node containing it is parsed whereas "user" would mean that it is evaluated when it is used by a query.

We add an xlink:actuate="noaction" alternative which does nothing: the unresolved link remains in the answer tree, but without the xlink:actuate="noaction" attribute. The intention here is that the application/user which stated the query should decide (and possibly pay) by himself when to finally resolve the XLink. Especially, information servers will use this alternative.

[1] in this work, the addressing part of XPointer is restricted to XPath expressions, returning a node set.

Activated Action: The activated action of XLinks is in some sense considered in the XLink working draft with the xlink:show attribute: the value "new" states that the target of the link is opened in a new window, whereas "parsed" states that the target of the link is inserted instead of the link.

In our context, there are more alternatives *what* to do:

- xlink:show="complete" parses the *whole* target *document* (useful in case that there are many XLinks to the document, such that it makes sense to regard it as a part of the database).
  When later a query traverses the link, it has only to evaluate the *xpointer* wrt. the already parsed document. Thus, xlink:show="complete" has to be accompanied by another specification how the actual query is answered.
- xlink:show="new" computes the result set of document(url)/$xpointer$ and keeps it somewhere in the database (useful if the same XLink occurs at several places).
- adopt xlink:show="parsed" for replacing the link by the result set of document(url)/$xpointer$ (i.e., materializing the answer as in Example 13.3).
- Additionally, xlink:show="virtual" means that the result set is computed and used (temporarily) for answering the query, but that it is not materialized.

Descendants strategy: The XLink working draft does not address this issue. We propose an additional xlink:descendants attribute that can be used to specify if the XLink should be transparent wrt. the descendant axis:

- the XLink itself may specify if it wants to be transparent wrt. the descendants relation,
- the application may specify which elements in virtual subtrees resulting from transparent XLinks are regarded as descendants:
  - "all",
  - "none",
  - "sameserver": those resulting from XLinks where the target url is on the same server as the source url,
  - "predefined": use the strategy given by the XLink.

The table given in Figure 13.1 shows which actions are necessary when a link is traversed the first time, and all subsequent times (note that exactly for the policy (auto, parsed), the links are later transparent, i.e., no additional action is required):

Note that different specifications for different XLinks in a document are allowed which can lead to an optimized differing behavior (e.g., when another XLink already accessed the complete document). It is even possible that another XLink has already precomputed the answer. Note that also a virtual answer may be cached if it is useful for another XLink (defined as (user, new) or (user, parsed)).

The combination xlink:show="complete new" provides a kind of double central caching when the target document is referenced with different XPointers, covering large parts of the target document (then the document is accessed only once by "complete"), and each of the XPointers occurs in several links (each of the answers is precomputed and stored once).

With xlink:actuate="auto", all XLinks in a subtree are processed when the subtree is parsed. This will potentially compute many answers which in fact are never needed. Applied recursively, xlink:actuate="auto" and xlink:show="complete" load transitively all documents which are reachable via XLinks (yet without evaluating individual pointers). This should only be done if it is known that the number and size of XML documents which are reachable transitively is suitably small, and that most of the XLinks will actually be queried. On the other hand, especially in the combination xlink:actuate="auto" and xlink:show="new" or xlink:show="parsed" (which precomputes and materializes all result sets of XLinks) the answering time for queries can be minimized.

	auto					user					
	cn	cp	cv	n	p	cn	cp	cv	n	p	v
transparent	–	xx	–	–	xx	–	-x	oo	–	-x	–
retrieve precomputed answer	xx	–	–	xx	–	?x	?-	oo	-x	?-	–
access doc	–	–	–	–	–	?-	?-	oo	?-	?-	xx
store document in cache	–	–	–	–	–	?-	?-	oo	–	–	??
compute answer from parsed doc	–	–	xx	–	–	?-	?-	oo	?-	?-	–
store answer in cache	–	–	–	–	–	?-	–	oo	?-	?-	??
replace link by answer	–	–	–	–	–	–	x-	oo	–	x-	–

o: makes no sense -: no action required x: action required

?: depends if the document is already cached and if the XPointer result is already computed by another link

1st entry: first time, 2nd entry: all subsequent times

Figure 13.1: Required Actions when Traversing an XLink

The combination xlink:show="complete virtual" specifies that for every query which uses the xlink, the answer is computed using the cached tree (saving space, but not redoing the Web access). The combination xlink:actuate="user" xlink:show="virtual" specifies that always the current information of the remote information source is queried (even if another XLink cached the complete document). In this case, the result set of an XLink has to be computed each time the link is traversed. Nevertheless, for large result sets, or for links which are not used frequently, this saves storage. The combination xlink:actuate="auto" xlink:show="virtual" makes only sense when xlink:show="complete" is also set (caching the complete document once and then answering the query from it).

When xlinks:descendants="yes" is used, the XLink must be considered whenever it is involved in a descendant navigation step. Thus, it is often preferable to materialize it, or to use additional metadata information to check if it may yield relevant answers (see below).

In practice, the services which provide the referenced documents constrain the possible alternatives:

"Hiding" Servers. If the referenced document is located on a server which does not publish the whole document, but only answers a (possibly even restricted) set of XPath queries, xlink:show="complete" is not applicable. The answer is always computed by the server.

"Lazy" Servers. If only a plain XML document (e.g., as ASCII) is provided, the client which initiated the query has to access the document and evaluate the XPointer expression by itself. All XLinks in the referenced document become – at least temporarily – references of local documents. Then, the client has to decide if he accepts the XLink strategies defined in the document. In case that the client specified xlink:show="virtual" for the link under consideration (assuming that the server will do the processing) he probably now changes his mind to materialize the answers (after having done the computation by himself).

If the client under consideration is an intermediate server, he probably decides not to resolve the link under consideration in this case, but to do xlink:actuate="noaction" and leave the resolving to his client.

Costs. Additionally, for the user of information services, the costs associated with an information source are relevant:

- xlink:show="complete" may be expensive when the costs depend on the size of data,
- xlink:show="new" or xlink:show="parsed" may be expensive when the costs depend on the number of queries.

- xlink:show="virtual" is potentially expensive in both cases. Since the result of resolving the XLink serves only as a base for evaluating another query which traverses the link, it is often cheaper to rewrite the XPointer expression given in the link together with the query to a new XPath query against the service (see below).

Information Providers. For an information provider, xlink:actuate="noaction" assures that he will not have any effort (neither computational resources, nor costs) for resolving the links in the documents he provides.

13.1.4 Optimizations

Resource Descriptions. If the provider of a referenced XML document also provides a *resource description*, it is often even possible to decide *whether* the actual query may be successful.

Especially, when XLinks are regarded as transparent wrt. the descendant axis, for expressions of the form .../*nodetest*, the simple information whether the linked document can contain elements which satisfy the *nodetest* can save the expense of querying the linked data source.

Example 13.4 (Subelements "through" XLinks)
The XPath expression

document("mondial-countries")//inflation

(which selects all inflation *elements by following the XLinks to all documents described above) does not need to follow the links to the* city *documents since these do not contain any* inflation *element.*

With a more detailed metadata description, e.g., in *RDF (Resource Description Framework)* [RDF00], the evaluation of many queries can be optimized.

Virtual Evaluation. If an XLink

<foo xlink:type=simple href=*url*#*xpointer* .../>

to an information service is specified as xlink:show="virtual", its result set is not materialized, but only used for evaluating another query which traverses the XLink. Let *remainder* be the part of the original XPath expression which remains when navigating from the node to which the query is applied to the xlink element. Then, instead of the original query, the rewritten query

xpointer/*remainder*

can be stated against the information service at *url* to answer the original query. In general, the result will be much smaller than when evaluating *xpointer* – and the computation is done at the server which is potentially optimized wrt. its database.

Example 13.5 (Rewriting Queries through XLinks)
In the query given in Example 13.2, the source mondial-memberships.xml *contains locator elements like*

<$\boxed{\text{country}}$ id="D" $\boxed{\text{xlink:href="document('countries.xml')/id('D')"}}$ />

which are used when answering the query

(∗) $capname IN $ms/@xlink:from→ $\boxed{\text{ms:country/@xlink:href}}$ ⇝ ctry:country/
ctry:capital/@xlink:href⇝cty:city/cty:name/text() .

For the above XPointer, the whole ctry:country *element with id "D" is returned, containing a* capital *subelement which is a simple link:*

```
<country car_code="D" ...>
    <capital xlink:href="document('cities-D.xml')#//city[@id='city-berlin']"/>
        ⋮
</country>
```

The ctry:country element is then queried by

ctry:capital/@xlink:href⇝cty:city/cty:name/text() .

which returns "Berlin". Instead, when resolving the first XLink, countries.xml the XPointer can already be extended with the remaining query, i.e., evaluating

document('countries.xml')/id('D')/ctry:capital/@xlink:href⇝cty:city/cty:name/text()

which will in turn query cities-D.xml with

city[id="city-berlin"]/cty:name/text()

which just returns "Berlin" which is also the answer to (∗)*.*

XML-Aware Web Caches. Another possibility is to use XML-aware cache technology in the Web which allows to store url-query-answer-tuples which occur frequently. This technology is investigated in [LM01].

13.2 XLinks in XPathLog

If an XPathLog engine interprets XLinks transparently, the user is not concerned with their handling: the underlying evaluation component silently evaluates the corresponding navigation step according to the behavior specified for the XLinks.

XPathLog rules can be used for prototypically investigating these strategies. Special built-in functions are used to provide Web access (cf. Section 15); a prospective extension implements the conversion and evaluation of XPath expressions given as strings into queries:

- *url*.parse@(xml): parses the XML document located at *url* and assigns its root to the reference *url*.parse@(xml), and
- *url*.evaluate@("*xpath-expr*",*Var*): evaluates the query *url*.parse@(xml)/*xpath-expr*→*Var*.

The following rules implement the xlink:actuate="auto", xlink:show="parsed" strategy:

```
X isa xlink :-       X[@xlink:type].
URL.parse@(xml),X[@targeturl→URL and @select→XPath] :-
                     X isa xlink[@xlink:href=XPointer], substring-before(XPointer,"#", URL),
                     substring-before(XPointer,"#",XPath).

<!-- handle simple links -->
 E[M→V] :-E/M→X, X isa xlink,
           X[@xlink:type="simple" and @targeturl→URL and @select→XPath],
           URL.evaluate(XPath,V).
<!-- handle locators -->
 E[M→V] :-E/M→X, X isa xlink,
           X[@xlink:type="locator" and @targeturl→URL and @select→XPath],
           URL.evaluate(XPath,V).
```

```
<!– handle arcs –>
 X[R→V] :-X isa xlink, X[@xlink:type="arc" and @to→Loc],
           X/sibling::*[@id→Loc and @xlink:role→R and
                        @targeturl→URL and @select→XPath],
           URL.evaluate(XPath,V).
 X[N→V] :-X isa xlink, X[@xlink:type="arc" and @to→Loc],
           X/sibling::*[@id→Loc and not @xlink:role and name()→N and
                        @targeturl→URL and @select→XPath],
           URL.evaluate(XPath,V).
```

The first rule alone implements the xlink:show="complete".

In all other cases, the evaluation component must (at least when a link is traversed the first time) be extended with built-in actions (as given in Figure 13.1).

14 XPATHLOG AND F-LOGIC: A COMPARISON

F-Logic [KLW95] is a *deductive object-oriented database language*. The experiences with F-Logic as a formal framework and as a language for data extraction and integration from the Web [LHL+98, May99a, MHLL99] provided the background for the design of XPath-Logic and XPathLog.

Already the syntax of XPath shows many similarities with the F-Logic syntax: complex expressions are built using navigation (in F-Logic written as country.population and country..city in contrast to the XPath notation using country/population and country/city[1]) and filters (written in both languages as *host*[*filter*]). The main conceptual difference is that the semantics of XPath expressions is given by result sets which are addressed by the expression, whereas F-Logic expressions return variable bindings which can also return items which are considered anywhere on the navigation path or in the filter.

Thus, the design of XPathLog as a crossbreed between XPath and F-Logic, combining the experiences with F-Logic as a successful (but "proprietary") language for data integration with the world-wide use of XML and XPath was a well-grounded evolution step. The following conceptual features tailored to data integration of XPathLog have been taken from F-Logic:

- property names as first-class citizens of the language (which e.g., can be equated to synonyms),
- variables at property positions,
- object fusion,
- class hierarchy with inheritance,
- lightweight signatures.

The implementation of XPathLog in the LoPiX system (see Section 15) is also based on the F-Logic system Florid.

14.1 F-Logic

F-Logic serves both as a logic and as a database querying and programming language (similar to the relationship between XPath-Logic and XPathLog).

The F-Logic data model is an object-oriented, semi-structured data model: the basic data model is based on *objects* which have properties (relationships between objects are regarded as object-valued properties). Objects are grouped by classes. The F-Logic data model is a *semistructured* data model since there is no fixed schema, but arbitrary properties can be associated with all objects. F-Logic supports class hierarchy, nonmonotonic inheritance, and signatures.

Since F-Logic strongly influenced the XPathLogic and XPathLog syntax, the presentation here can be kept short (for the full syntax and semantics of F-Logic, the reader is referred to [KLW95]):

[1] note that "." and ".." *both* correspond to "/" in XPath. Due to its non-tree data model, F-Logic has no direct equivalent to "//" – see below.

- The language is based on variables, constants, and *object constructors* from which *id-terms* are composed as usual. Id-terms are interpreted as elements of the universe. By convention, object constructors start with lowercase letters whereas variables start with uppercase ones. Ground id-terms play the role of *logical* object identifiers (*oids*).

In the sequel, let O, C, D, Q_i, S, S_i, Sc, and Mv stand for id-terms.

- An *is-a assertion* is an expression of the form $O : C$ (object O is a member of class C), or $C :: D$ (class C is a subclass of class D).

- The following are *object atoms*:

  - $O[Sc@(Q_1, \ldots, Q_k) \rightarrow S]$: applying the *scalar* method Sc with arguments $Q_1, \ldots, Q_k$ to O results in S,
  - $O[Mv@(Q_1, \ldots, Q_k) \twoheadrightarrow \{S_1, \ldots, S_n\}]$: applying the *multi-valued* method Mv with arguments $Q_1, \ldots, Q_k$ to O results in $S_1, \ldots, S_n$.

  Note that inside filters, the *property* position is always interpreted as an object, i.e., $\mathsf{o[m_1.m_2{\rightarrow}V]}$ is equivalent to $\mathsf{o[(m_1.m_2){\rightarrow}V]}$ and binds V to $\mathsf{o.(m_1.m_2)}$, not to $\mathsf{(o.m_1).m_2}$ as it would be in XPath and XPathLog; see also page 191.

- *Signature* atoms and atoms describing inheritable methods use the same syntax as defined for XPathLog in Sections 9.1 and 10.

- A *rule* is a logic rule $\mathsf{h} \leftarrow \mathsf{b}$ over F-Logic's atoms, i.e., is-a assertions and object atoms; a *program* is a set of rules.

Example 14.1 (F-Logic Database)
Below, a fragment of the database which is created by the application described in this paper is shown. For readability, we use mnemonic oid's of the form o_{name}.

o_{belg} isa country[name→"Belgium"; car_code→"B"; capital→$o_{brussels}$;
total_area→30510; population→10170241; continent@(o_{eur})→100;
indep@(date)→"04 10 1830"; pop_growth→#0.33; gdp_total→197000;
adm_divs↠{$o_{p\_antwerp}$, o_{p_westfl},...}; main_cities↠{$o_{brussels}$, $o_{antwerp}$,...};
ethnicgroups@("Fleming")→55; ethnicgroups@("Walloon")→33;
religions@("Roman Catholic")→75; religions@("Protestant")→25;
borders@(o_{france})→620; borders@($o_{germany}$)→167;
borders@($o_{luxembourg}$)→148; borders@($o_{netherlands}$)→450].

$o_{brussels}$ isa city[name→"Brussels"; country→o_{belg}; province→o_{p_westfl};
longitude→#4.35; latitude→#50.8; population@(95)→951580].

$o_{antwerp}$ isa city[name→"Antwerp"; country→o_{belg}; province→$o_{p\_antwerp}$;
population@(95)→459072; longitude→#4.23; latitude→#51.1].

$o_{p\_antwerp}$ isa prov[name→"Antwerp"; country→o_{belg}; capital→$o_{antwerp}$;
area→2867; population→1610695].

o_{p_westfl} isa prov[name→"West Flanders"; country→o_{belg}; capital→$o_{brussels}$;
area→3358; population→2253794].

o_{eu} isa org[abbrev→"EU";name→"European Union";establ@(date)→"07 02 1992";
seat→$o_{brussels}$; members@("member")↠{o_{belg}, o_{france},...};
members@("membership applicant")↠{$o_{hungary}$, $o_{slovakia}$,...}].

The basic F-Logic syntax and semantics has been extended in [FLU94] with *path expressions* in place of id-terms for navigating in the object-oriented model:

- The path expression O.M is *single-valued* and refers to the unique object S for which $O[M \rightarrow S]$ holds, whereas O..M is *multi-valued* and refers to every S_i such that $O[M \twoheadrightarrow \{S_i\}]$ holds.

Example 14.2 (F-Logic Path Expressions)
In our example,

$$o_{eu}.\mathsf{seat} = o_{brussels} \quad , \quad o_{eu}.\mathsf{seat.province} = o_{p\_westfl} \ .$$

The following query yields all names N of cities in Belgium:

?- C : country[name→"Belgium"], C..main_cities[name→N].

Since path expressions and F-Logic atoms may be arbitrarily nested into *molecules*, a concise and extremely flexible specification language for object properties is obtained – very similar to the XPath syntax. For the design (and also the implementation) of XPathLog, the experiences with the semantics of complex negated expressions in rule bodies, and the constructive semantics of complex expressions in rule heads which are in F-Logic both based on atomization were useful. Additionally, the F-Logic object algebra [Him94] which is implemented in FLORID has some similarities with the semantics defined for $\mathcal{SB}$ and $\mathcal{QB}$ in Section 6.2.

The semantics of F-Logic programs is defined by bottom-up evaluation, allowing for user-defined stratification.

14.2 Comparison

The basic F-Logic data model and querying language provides the following concepts which are not present in the XML/DOM data model:

1. the underlying data model for F-Logic (F-Structures) is a graph, all relationships are equally represented by object-valued properties.
2. there is an unsorted universe of objects which can simultaneously play the roles as objects, classes and properties. This allows for expressing complex correlations between the object level and the meta-level,
3. there are parameterized properties, e.g., *berlin*[population@(95)→3472009],
4. F-Logic distinguishes scalar (e.g., *germany*[capital→*berlin*]) and multivalued properties, (e.g., *germany*[city↠*berlin*]).
   Scalar properties have a special equating semantics: From the facts

   john[father→*paul*] and *john*[father→*mr_X*],

   the internal semantics derives that *paul* and *mr_X* are the same object (which has all properties of the objects *paul* and *mr_X*, similar as described in Section 11.6 for object fusion).
5. language: F-Logic allows variables to occur at arbitrary positions (i.e., especially, at property position).

Here, (3) and (4) are minor details of the modeling which are not supported directly in XML, but can easily be encoded:

ad 3) XML does not allow for parameterized properties. These can be encoded by attributes, e.g., the above example *berlin*[population@(95)→3472009], is represented by

```
<city id="cty-Germany-Berlin" country="D">
    <name>Berlin</name>
      <population year="95">3472009</population>
</city>
```

and queried by

?- //city[population[@year→Y]→P].

ad 4) The basic XML model does not distinguish scalar from multivalued properties; such constraints can be expressed in the DTD or XML Schema metadata specification. For XPathLog, the maintenance of scalarity by equating has to be encoded into suitable rules.

Example 14.3 (F-Logic: Maintaining Scalarity)
From a DTD or XML Schema metadata specification, information about cardinalities can be derived (extending the algorithms described in Section 10.2). Let

E.P[cardinality→C]

denote that P is a property of the element type E of cardinality $C \in \{$scalar, multivalued$\}$. Then, the following rule maintains this scalarity by fusing objects:

X = Y :- Z isa E, Z[P→X and P→Y] and E.P[cardinality→scalar]

where E is bound to a class name (i.e., in pure XML terminology, an element type) and P is bound to a property name (i.e., a subelement relationship name) and X, Y, and Z are bound to objects (element nodes).

The remaining differences (1), (2) and (5) directly effect the expressiveness and flexibility of the data model (graph vs. tree) and the language (variable bindings) and their combination (names and nodes vs. objects) and have been central "requirements" when designing XPath-Logic and XPathLog.

Tree vs. graph model. The main difference in the data model is that the F-Logic model is graph-based whereas the XML data model is tree-based.

- An F-Logic database consists of nodes which are connected by relationships. There is no additional structure in the graph, navigation is allowed starting at arbitrary objects and following arbitrary relationships. The graph-based F-Logic data model does not know a notion of distinguished root objects to generate a result (tree).
- The XML data model is a tree consisting of named nodes. The tree structure is defined by a distinguished subelement relationships. Additionally, reference attributes add subordinate cross edges to the tree. Literal contents and properties can either be represented by text nodes or by non-reference attributes.

  The XPath axes have no counterpart in F-Logic since in its non-hierarchical model there is no notion of children, parents, descendants, siblings etc.

The XPathLog data model combines the advantages of both models by distinguishing *virtual trees in a graph database*:

- the navigation graph distinguishes subelements from attributes and provides a principal hierarchical navigation structure which defines the tree axes, but also supports dereferencing of attributes and navigation along these minor navigation structures.
- the navigation graph allows for multiple parents, defining overlapping trees. This has been identified as a crucial feature for data integration (cf. Section 11).

Order. The F-Logic data model is unordered. With the handling of HTML trees in F-Logic [LHL+98, May99b], ordered trees have been *encoded* by using parameterized methods. Similar to the development of XML-QL from Strudel/StruQL and the migration of Lorel to XML (cf. Section 16), the DOM Herbrand structures augment the data model with order.

Variable Bindings. XPath does not allow to *bind* variables. Variable *references* of the form $*var* where *var* is bound by surrounding constructs (e.g., XSLT) are allowed at name positions and at value positions in XPath location paths. For data manipulation, binding variables in a querying clause and communicating them to an update clause has proven useful in many languages (also in the XML area, cf. XML-QL and Quilt/XQuery). Using the logic programming style semantics of declarative variable binding in expressions, XPathLog incorporates variable binding directly into XPath syntax.

Filter/Molecule expressions. As stated above, the syntax and semantics inside the "[...]" construct differs between F-Logic *specifications* and XPath/XPathLog's *filters*: F-Logic specifications specify properties of the host object, i.e.,

$\mathsf{o}[\mathsf{m}_1{\rightarrow}\mathsf{v}_1;\ \mathsf{m}_2{\twoheadrightarrow}\mathsf{v}_2;\ \ldots]$

describes the properties m_1, m_2 etc.; where m_2 is always interpreted as an object acting as a method: $\mathsf{o}[\mathsf{m}_1.\mathsf{m}_2{\rightarrow}\mathsf{V}]$ is equivalent with $\mathsf{o}[(\mathsf{m}_1.\mathsf{m}_2){\rightarrow}\mathsf{V}]$ and binds V to $\mathsf{o}.(\mathsf{m}_1.\mathsf{m}_2)$. E.g., the query

?- germany[city.name→N]   % F-Logic

results in false, since city.name is – in general – not defined. Here,

?- germany[city→_C[name→N]]   % F-Logic   (*)

binds N to the names of german cities.

In contrast, XPath/XPathLog's *filters* describe navigation in the subtree below the "host" element, navigation along paths is always binding to the left, including the host: $\mathsf{o}[\mathsf{m}_1/\mathsf{m}_2{\rightarrow}\mathsf{V}]$ binds V to $(\mathsf{o}/\mathsf{m}_1)/\mathsf{m}_2$.

From the XML-navigation point of view, the XPath semantics is preferable, whereas the F-Logic semantics allows for more complex databases, handling method names completely like objects, stating rules which select objects acting as methods in a complex way from the database. Deeply nested F-Logic atoms – especially using complex expressions at property position – can become very complex and powerful – but also very hard to understand.

The F-Logic evaluation component (which implements the object algebra [Him94]) is reused in the LoPiX system. Here, a rewriting component on parse-tree level has been added which maps relative location paths in filters to single steps as shown in the above (*) query.

Names as first-order citizens of the language. Whereas F-Logic uses a non-sorted universe where objects simultaneously act as objects, classes, and property names, the XML data model is sorted: the data is represented by elements and literals. The names of properties (element names and attribute names) are not part of the universe. Here, the XPath name() function (mapping a node to the element name, e.g., *berlin*/name() returns "city") can be used for relating property names with literals.

XPath does not allow to use the string resulting from the *name* method as nodetest in a query (this has to be encoded again in a filter, e.g. *path*/*[name()=$var] which makes such expressions much less readable).

On the other hand, with DTDs and XML Schema, strings occurring as data items are semantically directly connected with names, e.g., when defining enumerations in DTDs, or datatypes in XML Schema. Thus, any formalism which should incorporate the metadata information, e.g., for data integration, gains much from making names first-order citizens of the language:

Example 14.4 (Names as Data Items)
Recall Example 10.2 where the water types river, sea, *and* lake *have been used as element types, and as (enumeration) attribute values when describing the target watertype of a river:*

```
<!ELEMENT river (name,to?,...)>
```

```
<!ATTLIST river id ID #REQUIRED>
<!ELEMENT to EMPTY>
<!ATTLIST to watertype (river|sea|lake) #REQUIRED
              water IDREF #REQUIRED>
<!ELEMENT sea (name,...)>
<!ATTLIST sea id ID #REQUIRED>

<river id="river-rhein">
   <name>Rhein</name>
   <to watertype="sea" water="sea-north-sea">
</river>
<sea id="sea-north-sea">
   <name>North Sea</name>
</sea>
```

The following query returns all elements which violate this "integrity constraint":

```
XPath:     //river[to[not(@watertype = id(@water)/name())]]
XPathLog://river[to[not(@watertype = @water/name())]]→R.
```

Object Identity. Whereas F-Logic (and other object-oriented frameworks) explicitly use the notion of *object identity*, the XML data model does not use the term "object (or element) identity". The XML Query Data Model adds a concept of *node identity* to the basic XML model for handling references. The XML querying languages use – at least implicitly – an *internal* notion of object identity, e.g., for implementing the `id()` function.

The navigation graph data model also uses an *internal* notion of object identity. For data integration, object identity plays a major role when *fusing* and *linking* objects.

Further extensions. Additionally, XPathLog is extended with the concepts of class hierarchy, non-monotonic inheritance and signatures known from F-Logic as described in Sections 9 and 10. As shown in Section 12, the overall language is well-suited for nontrivial integration tasks.

14.3 Summary

The data model used by XPathLog for an *XML database* is in fact an XML-style "interpretation" of a semistructured (i.e., without having a fixed schema) object-oriented data model:

- the basic object-oriented model has been equipped with two types of properties: ordered subelements and unordered attributes (XML style),
- the data model is still a graph (non-XML style),
- there is a mechanism for defining *multiple, overlapping tree views* of the XML database.
- the navigational semantics of F-Logic has been adapted to the XPath style.

Thus, the data model *covers* the XML data model and embeds it into a more flexible data model for data integration and provides a powerful data manipulation language. The LoPiX system which is described in the subsequent section implements this framework.

15 THE LoPiX SYSTEM

XPathLog has been implemented in the LoPiX system [LoP] which extends the pure XPathLog language with a Web-aware environment and additional functionality for data integration.

15.1 Architecture

LoPiX has been developed using major components from the Florid system [FLO98, LHL+98], an implementation (in C++) of F-Logic [KLW95]. Due to the similarities between the F-Logic data model and the XML data model in general, and XPathLogic's multi-overlapping-tree model in particular, the Florid modules provided a solid base for an XPathLog implementation[1]. Especially, it was useful that the functionality of the complete module for the evaluation of a deductive language over a data model with complex objects could be reused. The system architecture of LoPiX is depicted in Figure 15.1 (which coincides with the Florid architecture except for the internal structure of *WebAccess*).

Storage. The actual (extensional) database is stored in the *ObjectManager*, the *ObjectManager-Access* provides a wrapper for the *ObjectManager* which is used by the *Evaluation* component.

The Florid *ObjectManager* implements a *frame-based* storage component which contains a frame for every object. A frame contains *slots* for storing properties of an object, including its class memberships and references to other objects. The frames are extensible to additional types of properties. For XML, additional kinds of properties have been added: *subelements* have to be distinguished from *attributes*, requiring the slot types for attribute data, attribute inheritance, and attribute signatures.

For LoPiX, the *ObjectManager* implements the DOM Herbrand model $\mathcal{HD}$ (note that this includes only the attribute and element axes), additionally, predicates, the class hierarchy, and signature atoms are stored. All elements of the Herbrand universe are represented by internal names (names and nodes by "artificial" ones – where we used mnemonic ids up to now, and literals "by themselves").

Data Model. The *ObjectManagerAccess* implements the abstract data model based on the database which is stored in the *ObjectManager*, i.e., the navigation graph extended with intensional properties (derived axes, transitivity of class hierarchy, downwards closure of signatures wrt. the class hierarchy, support for inheritance, object fusion, synonyms, built-in functionality for data conversion, string handling including matching regular expressions, arithmetics, aggregation operators, and annotated literals). It provides *iterator-based* declarative access to the database. The above intensional properties are not materialized, but implemented by the iterators. The LoPiX adaptations of the Florid *OMAccess* module were mainly concerned with the following aspects:

- extension of the querying interface to attributes and derived axes,
- define derived iterators for the derived axes based on the tree structure (induced by the subelement relationship) and on the child iterators (implementing the definition of $\mathcal{A}_{\mathcal{HD}}$ as defined in Definition 5.7). Here, it proved useful to reuse the handling of implicit transitivity of the class hierarchy also for the XML descendant relationship.

[1] all modules were subject to a profound reorganization of memory management and fixing several bugs which even survived extensive case-studies with Florid in the area of Web data extraction.

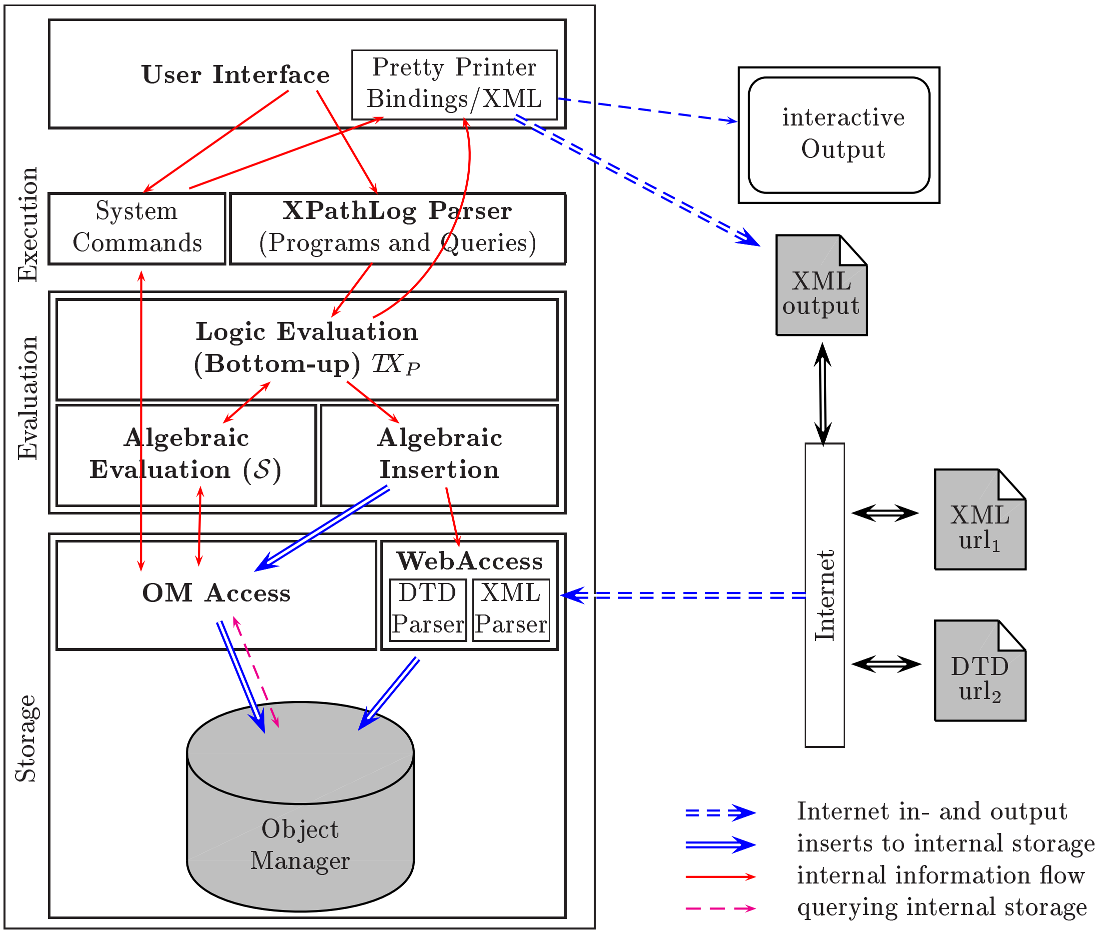

Figure 15.1: Architecture of the LoPiX System

- adaptation of insertion functionality to XML requirements.

The *OMAccess* is accessed declaratively from the *Evaluation* component by the following methods:

1. match($iname_1$,$iname_2$,rel)

   returns all pairs ($iname_1$, $iname_2$) such that $rel \in$ {member, subclass, descendant} holds between two internal names. member and subclass are concerned with the class hierarchy, whereas descendant is concerned with the XML tree. The relations are defined intensionally based on isa, subcl, and the child relation.

2. match([$iname_1, \ldots, iname_{arity}$], $arity$, $axis$, $methodtype$):

   For $arity = 3$, this returns all answers to the query ?- N[axis::Name⇝V]. For all XML axes, the default $methodtype$ is $mvData$ (multivalued data), because multiple results are allowed. For the attribute axis, additionally $inhData$ (inheritable data) is allowed, and for the child and attribute axes, also $mvSig$ and $inhSig$ (Signatures) are allowed.

   Additionally, there is $axis$ = noaxis which can be used for scalar data which is not part of the tree. Here, $arity > 3$ is allowed for *references* of the form $n.method@(arg_1, \ldots, arg_{arity-3}) \rightarrow v$ (as, e.g., in url.parse@(xml)).

3. match($pred$, [$iname_1, \ldots, iname_{arity}$])

returns the extension of a predicate $p(iname_1, \ldots, iname_{arity})$. Built-in predicates for comparison, equality, arithmetics, and string handling are also supported for this call. Built-in predicates have been extended for resolving *annotated literals* (cf. Section 5.6).

The *OMAccess* evaluates these queries wrt. the *ObjectManager*. If $iname_i$ is a constant, indexes are used for more efficient answering. Queries using derived axes are answered by first determining the nodes specified by the axis and then selecting those which match the given pattern. In detail, (2) implements the rules (3) and (4) of $\mathcal{SB}$ given in Definition 6.8: node[axis::nodetest→C] translates into

match([node,nodetest,*undef*], 3, axis, mvData) .

Answers are not materialized, but returned as an iterator (in the above case, iterating over the third position of the match pattern, i.e., replacing *undef* by ranging over possible values for C).

Insertions are handled analogously:

insert($iname_1$,$iname_2$,rel)
insert([$iname_1, \ldots, iname_{arity}$], $arity$, $axis$, $methodtype$)
insert($pred$, [$iname_1, \ldots, iname_{arity}$])

where for rel, only member and subclass are allowed (manipulating the class hierarchy) and for $axis$, only child, sibling, attribute, and noaxis are allowed. Equality is managed internally:

equalize($iname_1$,$iname_2$)

equates two internal identifiers (replacing $iname_1$ by $iname_2$ in the *ObjectManager* and maintaining a *synonym table*). The execution of inheritance steps is also implemented in the *OMAccess* which identifies an inheritable property and inserts the inherited fact (triggered by the *Evaluation* component according to the definitions given in Section 9.2).

In the subsequent section, it is shown that only the *OMAccess* (even, only the iterators in it) has to be adapted when complementing the proprietary *ObjectManager* with a native DOM Object Manager.

Web Access. The *WebAccess* functionality is closely intertwined with the *OMAccess* module. The FLORID *WebAccess* module implements the basic features of accessing sources via internet, and uses the SGML parser "SP" [Cla] for mapping SGML/HTML files to a proprietary tree representation in F-Logic [May00a].

Here, for LoPiX, additional parsing methods for mapping the XML tree to a representation of the navigation graph have been defined (cf. Section 15.3). Additionally, a method for mapping a DTD to XPathLog signature atoms based on the *lex/yacc* SGML DTD grammar [SM] has been implemented (cf. Section 10.2.1).

Evaluation. The central FLORID *Evaluation* module (*LogicEvaluation, AlgebraicEvaluation*, and *AlgebraicInsert*) provides in fact a *generic* implementation of a deductive language over a data model with complex objects.

LogicEvaluation implements a seminaive bottom-up evaluation of rules. *AlgebraicEvaluation* translates rule bodies and heads into the underlying object algebra and evaluates the generated algebraic expressions using the querying interface of *OMAccess*. The object algebra implements the semantics of XPathLog queries described in Section 6.2, generating sets of tuples of variable bindings. *AlgebraicInsert* instantiates the rule heads with the generated variable bindings and adds the corresponding facts into the database using again the *OMAccess* interface, implementing the semantics defined in Section 8. The evaluation of algebraic expressions does not materialize any intermediate result, but is purely based on nested iterators.

The whole evaluation module is actually independent from the underlying language and data model (assumed that the algebra itself and the mapping is given as a parameter).

Whereas in F-Logic, inserting a fact is idempotent (i.e., inserting it twice into the database did not matter), the generation of subelements by path expressions in XPathLog (cf. Section 7.2) is not idempotent. The *LogicEvaluation* component has been extended by bookkeeping about already inserted instantiations of every rule (cf. the dictionary extension to the $\mathcal{TX}_P$ operator in Definition 7.5).

Language Parsing. The *Parser* maps programs and queries into internal parse-trees which are processed by *LogicEvaluation*. The FLORID F-Logic parser has been replaced by an XPathLog parser. Exploiting the similarities between the F-Logic syntax and the XPathLog syntax, the XPathLog parser (i.e., its *lex/yacc* inputs) has been in fact derived from the F-Logic parser. An *XPathRewriter* module has been added for mapping XPathLog parse-trees to the F-Logic algebra before feeding them into the *Translator*.

Output. The *PrettyPrinter* outputs answers in the variable bindings format known from Prolog, or as an instantiation of the queries. Additionally, the result of queries which bind only a single variable can be output in XML ASCII representation.

The *export* functionality of the FLORID *PrettyPrinter* was very restricted, dumping the database contents in frame format. Here, LoPiX version is extended with an XML tree export function which exports the tree rooted in a given node according as a projection to a given signature as described in Section 11.1 (cf. Section 15.4).

UserInterface. The *UserInterface* module allows to use LoPiX from the command shell, including interactive queries and system commands. *SystemCommands* can also be executed in programs, mainly controlling program execution (user-defined stratification), debugging and formatting.

15.2 Dual-Memory Architecture

As described above, the *ObjectManager* stores the DOM Herbrand structure, predicates, the class hierarchy, and signature atoms in a frame-based model which is equipped with indexes for optimized access. The *OMAccess* encapsulates this model and adds some intensional closure properties. The proprietary *ObjectManager* module is not accessible by a native XPath interface. On the other hand, there are native, open-source DOM/XPath implementations available (e.g., Xerces/Xalan). The modular architecture of LoPiX allows to *combine* the original *ObjectManager* with a DOM implementation and an XPath interface as shown in Figure 15.2.

In the dual-memory architecture, the DOM stores the XML documents (i.e., it replaces the DOM Herbrand structure from the *ObjectManager*), whereas predicates, class hierarchy and signature atoms remain in the *ObjectManager (OM)*. Since the pure DOM model does not contain *any* index structures, also the tree structure (which implicitly defines the derived axes) and some index structures are stored in the OM.

As described in the previous section, the interface of *OMAccess* against *Evaluation* is completely declarative, accessing the *abstract* data model. The addition of a DOM storage component does not change the abstract data model. Thus, it does also not affect the interface.

For the dual-memory model, the internal logic of *OMAccess* has been adapted to be a mediator between the DOM/OM and the external queries. Nodes of the XML tree are stored in the DOM, but their class information, the derived axes, and the indexes are stored in the OM. The connection between both storage parts is provided by an additional data structure belonging to *OMAccess* for mapping OM internal identifiers to DOM nodes (implemented by two complementary dictionaries).

Queries on the child and attribute axes are mapped to queries against the DOM. As described in the previous section, queries using derived axes are answered by first determining the nodes specified by the axis and then selecting those which match the given pattern. Here, the first part of the task is still solved by querying the OM whereas the second part has to be answered by

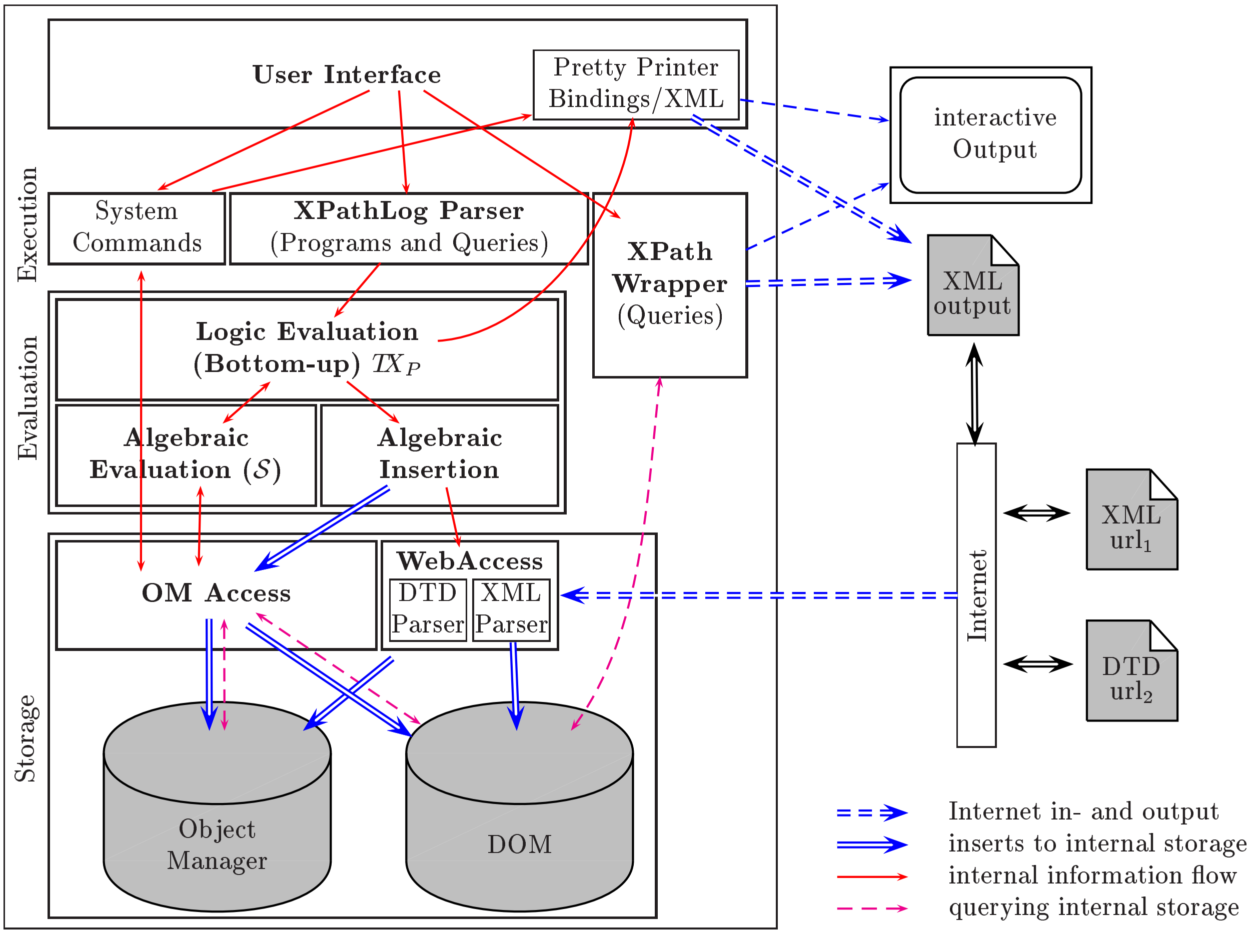

Figure 15.2: Dual-Memory Architecture of the LoPiX System

the DOM, connected by the OM/DOM identifier mapping. Similarly, insertions of children or attribute nodes are divided into a DOM and an OM part.

When parsing an XML source, the *WebAccess* module maps the tree into the DOM and maintains the class hierarchy and indexes in the OM with every step. When parsing a DTD, the signature information is added to the OM as before.

Problems. As already pointed out in Section 8.4, the DOM and XML Query data models require an element node to have a unique parent. Thus, data restructuring and integration by linking elements is not possible in the DOM model. Instead, elements (i.e., subtrees) have to be copied for generating a separate result tree, leading to the problems with reference attributes which have been described in Section 8.5.

Discussion. The dual-memory variant has been developed in [Beh01]. Here, the Florid architecture and design proved useful as a *generic* framework and “infrastructure” to implement a deductive language over a complex data model already for the non-DOM LoPiX variant. Even more, the seamless integration of an open-source native DOM storage module shows the flexibility and extensibility of the Florid/LoPiX modules as a base for deductive languages.

15.3 Accessing XML Documents

Every resource available in the Web has a unique address, called *Uniform Resource Locator* (URL), which is used to initiate access to the document. There is a predefined class url containing Web address strings. The class url provides a predefined built-in active method parse that, when applied to a member of url, accesses the corresponding Web document and adds it to the database. The signature for Web access is

(url subcl string)[parse ⇒ xmldoc].

Whenever for an instance u of class url the method u.parse(...) is called by a rule head, the document at the address u is automatically loaded and analyzed according to the arguments.

XML documents. u.parse@(xml) parses the document located at u as an XML document and generates the canonical DOM Herbrand structure as described in Definitions 5.5 and 6.4 is generated in the *ObjectManager*. The *document node* of this tree is associated with the *reference* u.parse@(xml). u.parse@(xml) has a unique child node which represents the outermost element node (e.g., the <mondial> element). Recall that multivalued attributes are split, and reference attributes are resolved.

Additionally, the extensions to a class hierarchy as described in Section 9 are added:

- every element type is made a class (subclass of object), and
- every element is made an instance of its class.

Internally, appropriate indexes and the dictionary for resolving annotated literals are added. This structure is then queried by XPathLog reference expressions.

Accessing DTDs. Similar to u.parse@(xml), u.parse@(dtd) accesses the document located at u as a DTD, applying the transformation described in Section 10.2.1 and generating suitable signature atoms.

Namespaces. As described in Section 11, when using several sources, it is recommended to introduce *namespaces*: When parsing an XML instance by u.parse@(xml,*namespace*), all names in the document are augmented with the given *namespace*. Analogously, u.parse@(dtd,*namespace*) creates the signature augmented with the given *namespace*.

Data-driven Web Access. By evaluating rules of the form

u isa url, u.parse@(xml) :- <body> ,

the internal database is extended by new XML documents. Thus, loading Web documents is completely data-driven. New documents are fetched depending on information and links (i.e., URLs or XLinks; cf. Section 13) found in already known documents.

As long as only a single document is considered, it is recommended to assign its (local) root to the global constant root which is used for evaluating expressions of the form "//...":

```
mondial[@xml→"file:smallmondial.xml" isa url].
U.parse@(xml) = root :- mondial[@xml→U].
```

If several documents have to be considered, it is useful to assign constants to their roots, e.g.

```
germany[@xml→"file:germany.xml" isa url].
france[@xml→"file:france.xml" isa url].
U.parse@(xml) = X :- X[@xml→U].
```

Then, the constants can be used as starting points for XPathLog reference expressions, e.g.

```
?- germany//city/name→C.
```

The above features are illustrated by the case study given in Section 12.

15.4 Exporting XML Documents

For every node N, the active method

N.export@(*doctype*,“*filename*”). % (in the head of a rule, or as a fact)

exports the XML ASCII representation of the subtree rooted in N, consisting of all subelements and attributes which are specified by the currently stored signature, to the given *filename*. The signature can either be given by parsing a DTD, processing an XML Schema instance (as described in Section 10.2.2), or as facts in the program. The DTD metadata (public/system, and the url) has to be set by methods, e.g.,

```
country isa doctype.
country.public = "mondial-europe-2.0.dtd".
?- sys.strat.doIt.
germany.export@(country,"exp1").
?- sys.strat.doIt.
```

If *filename* is the empty string, the output is sent to standard output.
The signature can be exported with the system command

?- sys.theOMAccess.export@(“sig”). or
?- sys.theOMAccess.export@(“sig”,“*filename*”).

When the result tree view is output, reference attributes are transformed back into `IDREF` attributes (generating artificial ids from the internal names of the objects/nodes if no ids are given).

15.5 Built-In Functionality

LoPiX supports built-in predicates and functions for comparison, equality, arithmetics, aggregation, and string handling including matching regular expressions and data conversion:

- *Comparison predicates*: “<”,“>”, “<=” or “>=”,
- *Arithmetic operations*: addition “+”, subtraction “-”, multiplication “*” and integer division “/”,
- *Aggregation*: *agg*{X[G_1,...,G_n];*body*} as described in Section 5.5.
- *String operations*:
  - string(arg) is true, if arg is a string.
  - strlen($string$, $value$) holds if $value$ (which can be given a constant or a variable) represents the length of $string$.
  - strcat($string_1$, $string_2$, $string_3$) succeeds if $string_3$ is the concatenation of $string_1$ and $string_2$.
  - substr($string_1$, $string_2$) holds if $string_1$ is a substring of $string_2$.
- *Regular expression matching*: pmatch($string$, *pattern*, *fmt-list*, *variable-list*)
- *Data Conversion*: string2integer(A,B), string2float(A,B), string2object(A,B)
- the above functionality also handles annotated literals.

For details, see [May00b]. LoPiX is available for Solaris and Linux at [LoP].

15.6 Performance Evaluations

Performance comparisons are still problematic in this area: the tools are incomparable, unstable, insufficiently documented, and sometimes simply too slow: Querying functionality can be compared with several XSLT implementations, XML-QL, the KWEELT Quilt implementation, and the commercial products eXcelon and Tamino. Although, the XPathLog query language is richer than these languages, allowing for dereferencing, joins between literals, and returning variable bindings.

Update performance is currently not comparable: Carrying out the MONDIAL data integration case study in XSLT would require a completely different program (according to the XSLT style which does not allow for incremental updates, but creates the result tree in a one-pass approach). Since XSLT does not allow for incremental updates, collecting a result view step-by-step, this task is far from easy, and presumably results in a much less readable program as the one given in Section 12 for XPathLog. As long as XML-QL and XQuery do not support updates, here also the tree must be generated in a single, large query. Due to their weak performance, the XML-QL and KWEELT systems are currently not suitable for such tasks. XQuery with updates (in case the problems with maintaining `IDREFS` described in Section 8.5 will be solved) would be suitable for data integration.

Tests have been run with a Sun Enterprise 350 workstation, 4 × 248 MHz processors, 1.6 GB memory (although, only one processor is used by each of the systems) using *top* for two queries:

1. “Select the names of all capitals of countries”,

2. “Select all names of cities which are seats of an organization and the capital of one of its members.” (cf. Example 3.4)

XSLT. Querying functionality has been compared with XT [Cla98][2]

- Parsing mondial-2.0.xml: about 8 seconds.

- capital names:

  id(//country/@capital)/name/text()

  Answering time (including XML parsing and validating) for `mondial-2.0.xml`: about 11 seconds.

- organizations as in (2):

  //organization[id(@seat) = id(id(members/@country)/@capital)]/abbrev

  Answering time (including XML parsing) for `mondial-2.0.xml`: about 17 seconds.

- cities as in (2):

  id(//organization[id(@seat) = id(id(members/@country)/@capital)]/@seat)/name/text()

  Answering time (including XML parsing) for `mondial-2.0.xml`: about 17 seconds.

[2] as the “winner” of several performance comparisons (e.g., by S.Rahtz in 2000 posted on `xsl-list`, at that time hosted by mulberrytech.com, and in March 2001 in a comparison given at `www.xml.com/pub/a/2001/03/28/xsltmark/index.html`). XT is also my personal favorite since it is stable, fast, and compatible with the standard – although (or because) it is a non-commercial experimental implementation since the first days of XSLT.

Excelon. The eXcelon [eXc] XML database system is under evaluation in [Weh01] in the PS3.0 beta version[3]. As a "full-fledged" XML database system, it allows for user-defined optimizations such as indexes. First experiments showed some instabilities (e.g., when trying to load a DTD; here, also the online-documentation of the betatest version is insufficient). After loading an XML instance, e.g., mondial-2.0.xml, into the database, the document can be queried, and indexes over attributes and elements can be defined. Queries which use either the descendant axis, joins, or dereferencing (as long as the DTD cannot be included, `IDREF` attributes can only be used in joins) show a poor performance.

- the simple query:

  ```
  /mondial/country/@capital
  ```

  needs some seconds.

- Selecting names of capitals by dereferencing via join:

  ```
  city[@id=/mondial/country/@capital]/name
  ```

  Answering time for `mondial-2.0.xml`: about 1:50 minutes, and about 10 seconds for `mondial-europe-2.0.xml`.

- Selecting seats of organizations which are also capitals:

  ```
  city[@id=/mondial/country/@capital][@id=//organization/@seat]/name
  ```

  Answering time for `mondial-2.0.xml`: about 7 minutes, and about 10 seconds for `mondial-europe-2.0.xml`.

  The slightly changed query (/mondial/organizations instead of //organizations)

  ```
  city[@id=/mondial/country/@capital][@id=/mondial/organization/@seat]/name
  ```

  needs about 2 minutes for `mondial-2.0.xml`.

Although eXcelon allows to define indexes, it seems that they are not used: With defining indexes on

- organization/@id (which should accelerate //organization and
- city/@id, country/@capital, and organization/@seat,

the above performance of the above query could not be changed. The product – at least the delivered beta version – is not convincing.

XML-QL. The performance and the stability of XML-QL [DFF+99c]. is not really satisfying. Especially, tests could only be run with the restricted database mondial-europe-2.0.xml. For the full database, XML-QL failed with a system error.

- Parsing mondial-europe-2.0.xml: 25 seconds.
- Capitals, first version: since XML patterns do not naturally support dereferencing, the first solution uses a join:

[3] Thanks to eXcelon Corp. for providing a betatest version.

```
function bla() {
    WHERE
    <mondial>
      <country capital=$cap></>
    </>
    IN "mondial-europe-2.0.xml",
    <mondial>
      <*>
        <city id=$cap>
          <name>$n</>
        </>
      </>
    </>
    IN "mondial-europe-2.0.xml"
    CONSTRUCT <capital>$n</>
}
```

Answering time (including XML parsing) for `mondial-europe-2.0.xml`: about 1:10 minutes.

- the same with XML-QL's dereferencing-by-subelement style (note that `@capital` is handled like a subelement in the XML pattern):

```
function bla() {
    WHERE
    <mondial>
      <country> <capital> <name>$n</></></>
    </>
    IN "mondial-europe-2.0.xml"
    CONSTRUCT <capital>$n</>
}
```

Answering time (including XML parsing) for `mondial-europe-2.0.xml`: about 25 seconds.

- seats, first version by join:

```
function bla() {
    WHERE
    <mondial>
      <organization seat=$seat> </>
    </>
    IN "mondial-europe-2.0.xml",
    <mondial>
      <country capital=$seat> </>
    </>
    IN "mondial-europe-2.0.xml",
    <mondial>
      <*>
        <city id=$seat>
          <name>$n</>
        </>
      </>
    </>
    IN "mondial-europe-2.0.xml"
    CONSTRUCT <city>$n</>
}
```

Answering time (including XML parsing) for `mondial-europe-2.0.xml`: about 1:20 minutes.

- the same with XML-QL's dereferencing-by-subelement pattern:

```
function bla() {
    WHERE
    <mondial>
      <organization seat=$seat> </>
    </>
    IN "mondial-europe-2.0.xml",
    <mondial>
      <country capital=$seat><capital><name>$n</></></>
    </>
    IN "mondial-europe-2.0.xml"
    CONSTRUCT <city>$n</>
}
```

Answering time (including XML parsing) for `mondial-europe-2.0.xml`: about 28 seconds.

Kweelt. The KWEELT implementation is *very* slow when navigating along references; note that the same tasks can be solved faster by joins.

- Parsing mondial-europe-2.0.xml: less than 5 seconds; parsing mondial-2.0.xml: less than 10 seconds.
- Selecting the ids of capitals:

```
<capitals>
   (
   FOR $s IN document("mondial-2.0.xml")/country/@capital
   RETURN $s
   )
</capitals>
```

Answering time (including XML parsing) for `mondial-2.0.xml`: about 10 seconds. Note that this query does not dereference the `IDREF` values.

- Selecting names of capitals by dereferencing the `IDREF`s of country/@capital:

```
<capitals>
   (
   FOR $s IN document("mondial-2.0.xml")/country/@capital->{city@id}/name/text()
   RETURN $s
   )
</capitals>
```

Answering time (including XML parsing) for `mondial-2.0.xml`: about 5 hours, and about 10 minutes for `mondial-europe-2.0.xml`.

- Selecting names of capitals by join:

```
<capitals>
   (
   FOR $id IN document("mondial-europe-2.0.xml")/country/@capital
   FOR $s IN document("mondial-europe-2.0.xml")//city[@id=$id]/name/text()
   RETURN $s
   )
</capitals>
```

Answering time (including XML parsing) for `mondial-2.0.xml`: about 50 minutes, and about 5 minutes for `mondial-europe-2.0.xml`.

- another query shows that the time does not very much depend on the size of the extension of the target class of the reference (which is given in the dereferencing hint):

```
<borders>
   (
   FOR $s IN document("mondial-2.0.xml")
                /country/border/@country->{country@car_code}/@car_code
   RETURN $s
   )
</borders>
```

Answering time (including XML parsing) for `mondial-2.0.xml`: about 70 minutes, and about 1:45 minutes for `mondial-europe-2.0.xml`.

The above results show that obviously indexes are missing in KWEELT. In the current state, the system cannot be used for non-trivial tasks.

LoPiX. Since LoPiX provides an interactive command-line interface, the answering time can be checked independent from parsing.

- Parsing mondial-europe-2.0.xml: about 35 seconds.
- Parsing mondial-2.0.xml and storing it in the *ObjectManager* needs about 3:50 minutes since it is not optimized for XML.
- Capitals:

```
?- mondial/country->C[name->N and @capital/name->CN].
```

Answering time for `mondial-2.0.xml`: less than 1 second processor time.

- Seats:

```
?- mondial/organization[@seat->C], mondial/country/@capital->C/name/text()->N.
```

Answering time for `mondial-2.0.xml`: less than 1 second processor time. Adding the check if the country where the seat is placed is actually a member by

```
?- mondial/organization[@seat->S and
                        members/@country->C[@capital->S/name/text()->N]].
```

requires about 8 seconds for `mondial-2.0.xml`.

Here, the existence of reference attributes and indexes proves useful. There are queries which take more time, e.g., searching for organizations which are seated in a country which is not a member:

```
?- /mondial/organization[abbrev->A and @seat[name->N]/@country->C
                  and not members/@country->C].
```

needs 6 seconds processor time. When the derived axes e.g., the descendants axis, are used such that more candidate nodes have to be tested, the anwering needs more time, e.g.,

```
?- //organization[abbrev->A and @seat[name->N]/@country->C
                  and not members/@country->C].
```

takes 15 seconds. There is still optimization potential in the evaluation and matching component. Although, as the above comparison shows, the answer times are tolerable for a research prototype, allowing for running and experimenting with the MONDIAL case study.

16 RELATED WORK AND CONCLUSION

16.1 Related Work

Much of the related work in the XML area has been presented in Section 3. Section 8 compared the pure XPathLog language with XML-Query, XSLT, XQuery, and related notions. In Section 14, in addition to the comparison of XPathLog with F-Logic, also some general comments and motivations on the data model have been stated. In Section 4.2 the data models used in other approaches have been analyzed, concerning the document vs. database duality.

In the sequel, a summary of approaches in Web querying, data integration, semistructured data and XML-related research is given.

16.1.1 Web Access, Web Querying, and Web Data Extraction

Although the present work does not focus on web access functionality, some related work is mentioned here since these languages and systems were the early predecessors of today's research in this area. They served for applying the *wrapper/mediator architecture* [Wie92] which has originally been designed for integration of heterogeneous databases to the upcoming "databases" on the Web (often as HTML pages). The early HTML web querying languages were not concerned with application-level semantics and integration of data, but served mainly as *wrapping languages* for Web contents. Some of them provide restructuring functionality within the internal data model.

As one of the first Web querying languages, **WebSQL** [MMM97] modeled the Web by two virtual relations which exactly correspond to documents and hyperlinks. For documents, attributes title, type, modif, length, and text – the latter containing the document source – are defined. Thus, these relations represented a kind of *Web skeleton*. WebSQL allows queries on the Web structure using this "virtual graph". The intra-document structure is not modeled, and therefore, cannot be queried either; only text search in documents using the `contains` predicate is possible. The SQL `SELECT FROM WHERE` syntax is extended with a `SUCH THAT` construct applied to every element of the `FROM`-clause to define the relevant portion of the Web to be materialized. The `SUCH THAT` condition is either an url or of the form `MENTIONS <string>` (which can be checked against an index); thus, the number of root documents for a query is finite. Web access is implemented by a *query engine* which – based on the `FROM` clause of the WebSQL query – either sends a request to an index server, or accesses the document tree specified by the WebSQL query via the http-protocol.

From the same group, **WebOQL** [AM98] additionally considers the internal structure of documents by making HTML parse-trees first-class citizens of the model that can be queried directly. Similar to our approach, they use a unified model of the inter-document level (Web) and the intra-document level (parse-trees). Browsing is regarded as a function (corresponding to our Web interface method parse) that maps urls to *hyper-trees*, or *"webs"*. Thus, all HTML pages are handled by a generic built-in interface that maps an HTML page to a hyper-tree without requiring external wrappers. WebOQL queries are then performed against the current web; additionally, new webs can be built.

One important distinction between WebOQL and current the XML data model is in the "weight" of the data model: while the XML data model is a "lightweight" data model (i.e., there is almost no fixed schema), the WebOQL hypertrees are based on a fixed generic tree schema.

Similar to OQL, WebOQL is a purely functional language with additional operators for manipulating hypertrees; the results of WebOQL queries are always hypertrees.

W3QL/W3QS [KS95,KS98] is another early Web querying language/system. The basic data model is rather simple and contains only the Web structure: The Web is mapped to a graph consisting of nodes (Web documents) and links (hyperlinks). For HTML documents themselves, only specific tags are evaluated, such as title, title.content, content-length, anchor[i] (the ith outgoing hyperlink) – (similar to those which are extracted automatically by FLORID's url.get method). The syntax of W3QL is similar to SQL, with the `FROM` clause augmented to handle paths in the Web graph, and the `WHERE` clause augmented to state conditions on the variables denoting Web pages and hyperlinks. In

```
SELECT ... FROM n1, l1, n2 WHERE ...
```

n1 and n2 have to be bound with urls, and l1 has to be bound with a hyperlink from n1 to n2. Special types of general path expressions are also allowed in the `FROM` clause. The `WHERE` clause must contain a *domain condition* of the form n1 in <set-of-urls>, specifying a finite set of urls to be the roots of the search. In contrast to the other described approaches, all other conditions in the `WHERE` clause (concerning the above-mentioned document properties) are not evaluated against an internal data model, but by external W3QL-specific programs (e.g., `SQLcond` and `Perlcond`):

```
WHERE n1,n2: SQLCOND '(n1.format = HTML) AND
                      (n2.title = "Information Systems")'
WHERE l3,n4: PERLCOND
    '(l3.content =~ /href\s*=\s*"(.*)"/i) &&
     (n4.content =~ /$1/)'
```

Thus, in W3QL/W3QS, the facilities for querying the document structure are very restricted.

In contrast to the other Web querying approaches, W3QS has a non-integrated architecture using external tools. Thus, it can be easily extended to additional file types (LaTeX, BibTeX, postscript), and to more sophisticated condition-evaluating programs.

WebLog [LSS96a] is a deductive Web querying language operating in an integrated graph-based framework. Although its syntax resembles F-Logic [KLW95], it is not fully object-oriented; the only objects are *"rel-infons"* which simulate in some sense the nodes of the parse-tree. Using rel-infons, atomic formulas describing HTML pages are of the form

<url>[<rel-infon-id>: <attr>→<val>] ,

that describe properties of a certain rel-infon on a Web page. Although, the data model does not imply a direct mapping of parse-trees to rel-infons – the actual mapping is left to the user. It it is left open whether or not this can be done in a generic way. Built-in predicates hlink and htext, define the semantics of hyperlinks. Thus, the actual data model is somewhere between the Web skeleton and the extended Web skeleton. Navigation along hyperlinks is provided by the above built-ins, but navigation in the parse-trees is not supported. Since the parse-tree is only partially represented, the evaluation of pages is mainly based on substring matching. There is no reported implementation of WebLog.

The above Web querying languages do not deal with the semantical analysis of document contents, wrapping, or restructuring of information. Although, especially in WebOQL, wrapping in a similar style as in our approach by generic queries should be possible.

Jedi [HFAN98] is a tool for manually specifying Web access and wrappers for HTML pages by combining grammars and rules. **W4F** [SA99] is a toolkit for interactively generating wrappers for HTML pages using HEL, a DOM-based language operating on the parse-tree of a document. Here, comma-lists and name-value pairs are regarded as atomic – i.e., the result is not completely mapped to the application domain, requiring a manually specified application-specific postprocessing. HEL focusses on wrapping single Web pages; Web exploration and information integration is left to the application. Another tool for interactive generation of wrappers using the DOM model is presented in **NoDoSE** [Ade98]. [KWD97] present inductive learning methods for wrapping HTML pages with a tabular layout. [ECY+98] present a tool for automatical matching-based wrapper generation for *data-rich and ontological narrow* sources.

To complete the list of wrapping languages, F-Logic has been used in the Florid system for wrapping HTML sources [LHL+98, MHLL99]; e.g., the original Mondial HTML sources have been transformed into XML by using Florid [May99a].

Prolog-style (i.e., with cut and fail semantics), and Constraint Logic Programming approaches for programming Web access and generation of HTML documents have been presented in [HC96, Tar99]. Here, in contrast to the intuitive bottom-up semantics, the operational semantics of Prolog evaluation is exploited in combination with imperative programming languages.

16.1.2 Semi-Structured Data and Data Integration

As already mentioned in Section 2, research on *semistructured data* since the mid 90s strongly influenced the development of XML. Early approaches to semi-structured data, especially focusing on semi-structured data as databases (in contrast to documents) were OEM/Lorel [GMPQ+97], Strudel/StruQL [FFLS97, FFK+98], UnQL [BDHS96], and F-Logic [KLW95, LHL+98], using "proprietary" semi-structured data models of the respective languages (in pre-XML times). The focus of all these approaches was on mediation and data integration. With these, also Logic Programming style languages have been used for manipulating and integrating semi-structured data(bases).

The **Tsimmis** project [GMPQ+97] is a "typical" implementation of the *wrapper/mediator architecture* [Wie92] for integration of heterogeneous databases, now also including Web databases. Tsimmis uses *OEM* (Object Exchange Model) as a common data model for the extracted, application-level data. An OEM database consists of objects and subobjects without any fixed schema. The OEM data model is an edge-labeled graph, which also has a textual representation:

Example 16.1 (OEM)
An excerpt of the Mondial *database in OEM looks as follows:*

```
<&cont1, continent, set, {&a1, &n1, &c1, &c2, &c3, ...}>
<&n1, name, string, 'Europe'>
<&a1, area, number, 9562488>
<&c1, country, set, {cn1, cc1, ca1, cp1, cap1, ctys1, n1}>
<&c2, country, set, {cn2, cc2, ca2, cp2, cap2, ctys2, n2}>
<&cn1, name, string, 'Germany'>
<&cc1, code, string, 'D'>
<&ca1, area, number, 356910>
<&cp1, population, number, 83536115>
<&cap1, name, string, 'Berlin'>
<&ctys1, cities, set, {cty1, cty2, ...}>
<&n1, neighbor, set, {c2, c3, ...}>
<&cn2, name, string, 'Belgium'>
```

In Tsimmis, the Web structure and the page markup is not modeled. Instead, it is presumed that external *wrappers* are given that map Web pages to the OEM representation. Different languages are used for wrapper and mediator specification (MSL and WSL, respectively), and for querying (MSL and Lorel). In the wrapper specification language WSL, rules specify which action has to be executed on the resource for a given query, e.g., (possibly using external resources) to yield OEM atoms. E.g., for a relational database or for a query form, rules look as follows:

```
C :- <C country {<code $CODE>}>
   //$$ := "select name
            from country
            where code=$CODE"//

C :- <C country {<code $CODE>}>
   //$$ := "find-code $CODE"//
```

For ASCII or HTML sources, an additional external, non-WSL wrapper layer (e.g., by perl scripts) would be necessary. MSL maps OEM data to OEM views, similar to F-Logic rules. E.g., all cities which are stored as capital (names) in *CIA*, and are also stored in *GlobalStatistics*, are collected by the MSL rule

```
<capital {<name Cap> <country CN> R }>@med :-
<country {<name CN> <capital Cap>}>@cia
AND <country {<name CN> <city {<name Cap> | R}>}>@gs
```

The MedMaker tool [PGMU96] implements MSL as part of the Tsimmis project.

The **Lorel** language [AQM+97b] for querying semistructured data uses the graph-based OEM data model. In OEM, data is represented in a graph whose nodes are objects, and whose edges are labeled with attribute names. Complex objects are simply collections of objects; leaf nodes have atomic values. Lorel provides SQL/OQL-like constructs, extended by powerful *general path expressions* which are regular expressions for characterizing paths in an OEM model. E.g., the following query selects all cities in countries which are reachable by land from Germany:

```
select X.city.name
from Mondial.country X, Mondial.country.Y
where Y.name = 'Germany'
  and Y.(border)+ = X
```

Lorel has been implemented in the Lore system [MAG+97]. In contrast to Web query languages/systems, Lore is not tailored to deal with Web pages – the OEM data is assumed to be provided by source-specific wrappers that can be implemented in any language providing an OEM interface, especially, the Tsimmis (described below) wrappers can be used. Lore has been migrated to XML in [GMW99] (see below).

In the **Strudel** system [FFLS97,FFK+98], the Web is also mapped to a unified internal, OEM-like graph representation. The language StruQL is used both as a query and a transformation language on this graph model. The StruQL language is based on a `WHERE ... COLLECT ...` statement. The `WHERE` part specifies conditions that determine which objects will be selected. Similar to Lorel and UnQL, regular path expressions are allowed. During the evaluation, the `WHERE` clause generates all variable bindings. The `COLLECT` clause places the results in collections for further use (or returns them as the result of a query). The syntax is extended for Web page generation with `CREATE` and `LINK` commands which generate nodes and links from the variable bindings. Wrappers to the internal Strudel graph model have to be provided independently. Strudel/StruQL provided the base for XML-QL [DFF+98,DFF+99b] (see Section 3.5) and its implementation, which builts on top of Strudel: XML-QL queries are translated to StruQL queries.

A similar light-weight data model is used in [AV97], with the main difference that the children of each vertex are ordered. Integration based on (re)construction of data forests is done by declarative rules, using *tree terms*.

UnQL [BDHS96] also uses an edge-labeled graph-based model for semistructured data and defines a language for navigating and querying it. UnQL models can be restricted by *graph schemata* that constrain the allowed paths in the graph. Graph schemata can be used for query planning and optimization. UnQL queries also have an OQL-like syntax of the form `SELECT ... WHERE ...`; instead of a `FROM` clause, reference expressions starting from the root node and other distinguished nodes are given in the `SELECT` part. Additionally, *restructuring queries* to construct new trees are possible. Here, *structural recursion* is used. In contrast to the classical `SELECT ... FROM ... WHERE`, this allows to express queries and data transformations also in the presence of cycles in the graphs.

Other early projects were *Garlic* [CHS+95, RAH+96] for integration of heterogeneous sources using an adaptation of the ODMG data model, and *Disco* [TRV96].

An interesting aspect here is that all of the above approaches used a graph-based data model with only one type of relationships (cf. also Section 4.2).

As a non-graph based approach, the **Araneus** project [AMM97] uses a hypertext-based model for Web site restructuring. This system uses different languages to extract data and define views on it.

For a complete overview of these early systems, see [FLM98]. In the meantime, SGML had moved into the focus as a semistructured data format.

There are some more approaches to semistructured data which also apply to XML, but do not use the usual XML-relates concepts (such as XPath or XML patterns).

The **YATL** language of the **YAT** system [CDSS99] is a pre-XML proposal, already using SGML and DTDs. Its trees provide a unified model for relational, object-oriented (ODMG), and semistructured/document data (SGML). The YATL language follows a pre-XML-style rule-based design for complex objects in the style of MSL or F-Logic. In [CCS00], the YAT system is turned into an XML system for data integration. Although YAT uses the XML data model, YATL does not use any XML/XPath language constructs. After mapping an XML instance to a YAT tree, there is no notion of attributes. Dereferencing is not explicitly supported, and it has no notion of the XML axes (similar to the same issue for XML-QL); instead it supports regular path expressions and tree algebraic operations. Thus, it is not directly comparable.

XDuce ("transduce") [HP00] is a functional-style tree transformation language which is based on *regular expression pattern matching*. Here, regular expressions are an extension of (originally, SGML) DTDs which are used to formulate queries against XML instances. Based on the pattern, the language also associates a *type* with every query.

TQL (Tree Query Language) [CG01] is a proposal for querying trees based on *ambient logic*, a modal logic.

16.1.3 XML & friends

Several XML querying and transformation languages have been presented in Section 3.

Data Model and Languages.

First implications on the data model considering different requirements for documents and databases have been stated in Section 4.2; concluding that ordered and unordered versions of data models and languages are reasonable from different points of view. The pure XPathLog has been compared with the common XML languages and notions in Section 8.

Systems and Products.

Additional to and around these "basic" XML languages, there are several projects, dealing with the design of XML environments and data integration, up to commercial B2B systems.

KWEELT [Sah00] (see also Section 3.10) implements Quilt over a standard DOM implementation. As a "pure" experimental implementation of a language, it is comparable to LoPiX (which also does not claim to be a "product"). For a comparison of the systems see Section 15.6.

XML-GL [CCD+99] is a graphical language for handling XML data in a "Query by Example" style. XML-GL introduces an explicit *XML Graphical Data Model* XML-GDM which represents DTDs and XML instances. XML instances are represented by node-labeled trees where attributes and text contents are represented as leaves. The DTD uses the same representation, annotated with cardinalities and contents model information – yielding a generic pattern of the valid instances. The language is directly based on this representation, based on *extract-match-clip-construct-queries*; the tasks of the individual steps are closely related to those in XML-QL or XQuery: The *extract* part identifies the scope of the query, the *match* part states additional conditions which elements are relevant, the *clip* part accesses subelements and attributes of selected items, and the *construct* part generates the result.

LORE [MAG+97] has been migrated to XML in [GMW99]. In several aspects, the migration is similar to what has done from FLORID to LoPiX: the original data model and the language have been extended with the XML distinction between attributes and subelements.

The originally used OEM model has been adapted to the special properties of XML: An XML element is a pair $(eid, value)$, consisting of an ID and a value which again consists of four elements: a *tag*, an ordered list of attribute-value pairs, a list of *crosslinks*, representing reference attributes, and an ordered list of children in the form of $(label, eid)$. Here, reference attributes are stored twice: in a *semantic* representation as links, and in a *literal* representation as text-valued `IDREF(S)` attributes.

The language is also adapted. The original LOREL language already supported path navigation, thus only the selection between elements and attributes had to be added: expressions of the form mondial/>country/>city navigate along the subelement relationship, whereas in mondial/>country/@car_code, the second step selects an attribute. In contrast to the XPathLog/LoPiX migration LOREL does not support the other XML axes, but retains the regular expressions from the original LOREL language. Comparisons are also augmented in the way described for *annotated literals* in Section 5.6. Additionally, *range qualifiers* and *skolem functions* similar to MSL and STRUQL/XML-QL and YATL are introduced.

The MIX *(Mediation in XML)* system [BGL+99] uses the XMAS *(XML Matching and Structuring)* language, derived from XML-QL for data integration in an architecture which has been influenced by the mediator architecture [Wie92] of TSIMMIS. MIX regards XML as a database model instead of a document model. An XML view is defined in XMAS by the mediator administrator over one or more data sources. The sources themselves are not necessarily in XML; here wrappers from the TSIMMIS project can be employed. Additional to the XMAS querying language, a graphical user interface, called BBQ, similar to XML-GL is provided. In [PV99] it is described how to derive the DTD for a given XMAS view definition.

The SILKROUTE system [FTS00] is closely related to the XML-QL project. The main goal of the project is the automatical conversion from relational data to XML, conforming to arbitrary DTDs. Applications express their queries in XML against the view provided by SILKROUTE. Here, XML-QL is extended to *RXL (Relational to XML Transformation Language)*. On the relational side, RXL has the full power of SQL, and on the XML side it has the full power of XML-QL: RXL queries are of the form

FROM *relations*
WHERE *sql-condition*
CONSTRUCT *xml-pattern*

Similar to SQL, the `FROM` clause binds variables which are then used in the `WHERE` and `CONSTRUCT` clause. The syntax of `CONSTRUCT` clause is the same as for XML-QL, except that nested queries are again of the `FROM - WHERE - CONSTRUCT` form. Typical RXL queries are complex since they *map* an external XML-QL query to the underying relational database using the predefined views. Note that not the view is an RXL query, but the incoming XML query is composed *with* the RXL query representing the view to the actual query against the database.

Software AG's TAMINO [Sof] is a commercial XML platform for electronic business, consisting of storage, development, and integration components for XML data and applications. Originally, TAMINO has been based on a hierarchical database model. The pure querying interface was first XQL-based, and has then be adapted to the XPath standard (e.g., accessible by queries consisting of the server url, and an XPath expression). Tools are provided for creating and manipulating

XML documents, DTDs and XSL stylesheets. For building applications (e.g., B2B, document management, electronic publishing), Java APIs are supported.

eXcelon [eXc] is another XML-based B2B product (originally based on ObjectDesign's ObjectStore platform). The basic XML server provides a direct querying interface for XPath (previously, for XQL), and for XSLT stylesheets. The pure XSL language is extended with updates by *XUL (XML update language)*. Several Java APIs are provided (e.g., DOM Level 2) for application development; additionally, B2B packages are available.

In the meantime, both products claim to use a "native" XML storage – whatever this means. Both products include Web server functionality and support data exchange with relational database and ERP products. The XML storage section is also designed in a large-scale database-system style: the systems are able to store many individual documents persistently, maintaining and applying access privileges, indexes etc.

The paper [TIHW01] does not only address updates in native XML languages, but also presents algorithms for maintaining XML repositories based on relational database systems. In their opinion, "most frequently-updated data tends to reside in relational systems". (Recall that the SilkRoute project above deals with similar issues.) Thus, well-studied techniques for updating complex XML structures which are mapped to relational systems are required.

The mapping between the relational data model and the XML data model has already been investigated in several projects. In SilkRoute [FTS00] and Xperanto [CFI+00], an XML mediator is placed over an (object-)relational database management system. In SilkRoute, an SQL-to-XML view definition is given by the user. Then, XML queries against this view are translated into SQL and submitted to the underlying database. In Xperanto, the underlying database is mapped to a default XML view, and users define their private views wrt. this view. Internally, these views are translated into SQL queries.

Thus, the relational data is actually not mapped to XML, but XML *queries* are translated into relational ones over a given, fixed relational schema. In contrast, [DFS00, SGT+99, FK99] deal with XML databases over a relational core: the original schema is given in XML, and a suitable (optimized) relational schema is generated from the input (and the input data is stored accordingly). Here, two principal approaches emerged:

[FK99] prefer the *edge* and *attribute* approaches which both implement the idea of a *universal relation*: In the *edge* approach, each element or attribute instance is stored as a tuple in a large "edge" relation. In the *attribute* approach, each tag or attribute name defines an individual binary relation. The disadvantage is that each element distributes over many tuples (and also relations in the *attribute* approach) which requires large joins for evaluating queries. On the other hand, by using a universal relation, the approach can also handle XML documents which do not have a DTD or XML Schema description which could be used to generate an optimized schema.

In contrast, [DFS00, SGT+99] use the *shared inlining* method which tries to store as much data of an element in a single tuple as possible. The main idea is the same as when deriving a relational schema from an ER diagram: 1:1 or 1:n-relationships can be stored together with the entity information on the 1-side. In the XML case, all subelement names which occur at most once for an element, and all attributes can be stored with their element. Note that (in contrast to the X-structures underlying the XPathLog/LoPiX data model) most approaches regard `NMTOKENS` and `IDREFS` attributes as scalar which are split and resolved on-demand. Every such "element" tuple also has an (internal) `ID` and a `parentID` attribute to store the tree structure (here again, it must be decided if an ordered or unordered model is implemented).

[TIHW01] follows the *shared inlining* method and presents algorithms for translating updates on the XML repository to its relational representation in the database. In addition to defining language-independent update operations (see Section 3.12); these operations are incorporated into XQuery, which is then used for evaluating the performance of the presented update algorithms. The problem of handling (i.e., splitting and resolving) `IDREFS` is still left with XQuery. Similarly, the problem of dangling references is addressed in the paper; since XQuery allows for dangling `IDREF` attributes, they are also allowed for updates.

Naturally, implementations of XML repositories over relational cores come also from the well-known RDBMS vendors: Oracle's *9i ApplicationServer* supports XML data exchange and Internet communications; expecially, the XSQL package defines a fixed, canonical mapping from relational data to XML. IBM's *DB2 XML Extender* does the same, additionally providing *DADs (Data Access Definitions)* for generating XML according to user-defined DTDs from the relational data. Similar products are available for Microsoft SQL server.

Observation: Data Models.

The presented systems use several underlying storage mechanisms. TAMINO and eXcelon claim to use a (proprietary) "native" XML storage. LOREL migrated the TSIMMIS OEM model to XML. The "proprietary" XML-QL data model inherited from STRUDEL/STRUQL is used by the MIX project. In contrast, the XML-QL-based SILKROUTE system does *not* use the XML-QL data model, but applies the XML-QL language only as a language over abstract views, mapping the actual queries to SQL queries against an underlying (object-)relational database system. The "mainstream" (RDBMS vendors (naturally), but also the authors of [TIHW01] who developed W3C XQuery and the W3C XML Query Data Model) also uses XML only as an intermediate data model to the user and for data exchange, whereas the actual storage is a relational system.

There is (yet) no "serious" system which actually uses a DOM model and implementation for storage. Only the research systems KWEELT and the dual-memory variant of LOPiX use a DOM implementation (although, due to the linking/copying problems described in Section 8.5, the proprietary *ObjectManager* variant of LOPiX seems to be preferable).

The data models used by the above systems differ in characteristic properties:

- graph vs. tree
- edge-labeled vs. node-labeled
- node/object/element identity

Graph vs. tree. As already stated above, the approaches which have their roots in the pre-XML time, i.e., OEM/LORE, STRUDEL/XML-QL, and F-Logic/XPathLog adapted graph models to the XML specialties:

- subelement/attribute duality: the graph models knew only a single kind of relationship. The data models have been extended appropriately with $(name, value)$ pairs (where XPathLog silently splits `NMTOKENS` and `IRDEFS` attributes), and the languages have been extended. For XML-QL, dereferencing in the querying part is weak. Here, LORE provides a special feature by allowing for a *semantic* and a *literal* view of `IDREF(S)` attributes.
- language adaptations: XPathLog uses the XPath syntax for selecting navigation axes. XML-QL uses XML patterns, LOREL extends its original language – both do not support XML axes, but regular path expressions.
- ordered nodes/children: XML-QL uses a global order of elements whereas LORE and XPathLog use a local order of children.

YAT/YATL is originally tree-based, but does not completely support XML notions such as attributes, axes, and dereferencing.

Edge-labeled vs. node-labeled. The data models which originated from graph-based data models (LORE, XML-QL, and XPathLog/LOPiX) are edge-labeled. As long as only trees are regarded, it does not matter whether a data model is edge-labeled or node-labeled. But, when – e.g., in case of data integration as shown in Section 11 – an element is allowed to have multiple parents, an edge-labeled approach is preferable to represent different child relationships wrt. the same element. An interesting feature of the XML migration of LORE is that it is a mixture of a node-labeled and an edge-labeled model: each node has a tag, *and*, additionally, the relationship is labeled with a name. This especially supports XML Schema where *element types* are distinguished from the names of the child relationships. The XPathLog extension with classes supports similar functionality: relationships are labeled with *element names*, and elements can be members of classes.

The W3C XML data models, i.e., the DOM and XML Query data model, are node-labeled trees, with the restrictive consequences wrt. data integration described in Section 8.5.

Node/object/element identity. The data models which originated from graph-based data models (and from the object-oriented area) use object identity as a natural feature. The XML data models do *not* know about any modeling concept like "element identity" – nevertheless, they also have to use it for internal storage. Explicit element identity in form of skolem functions is especially used in XML-QL and the LOREL XML migration.

Implementations and Performance. Performance comparisons are still problematic in this area: the tools are incomparable, unstable, insufficiently documented[1], and sometimes simply too slow. Querying functionality can be compared with several XSLT implementations, XML-QL, the KWEELT Quilt implementation, and the commercial products eXcelon and Tamino. Although, the XPathLog language is richer than these languages, allowing for dereferencing, joins between literals, and returning variable bindings. Here, the XSLT tools which benefit from using an "established" language (tested: XT) showed the best and most stable performance for answering XPath queries. As Tamino and Excelon are "full-fledged" XML database systems, their querying performance heavily depends on user-defined optimizations such as indexes.

The other systems, i.e., XML-QL, KWEELT, and LOPiX are "research systems". The performance of XML-QL is not really convincing, but acceptable. KWEELT is quite slow, it is obvious that indexes are still missing. The performance of LOPiX is much better than KWEELT, and also better than that of the XML-QL implementation – here it shows that LOPiX profits from the storage management and optimization techniques which are "inherited" from FLORID (which are partially also used in the DOM-based dual-memory version):

The original LOPiX version builds upon the frame-based *ObjectManager* storage module of FLORID which provides some optimizations using indexing (which are also used by the dual-memory variant). The answer times are tolerable for a research prototype, allowing for running and experimenting with the MONDIAL case study.

In contrast to KWEELT, the DOM of the dual-memory LOPiX is complemented by indexes residing in the OM, which leads to an acceptable performance for queries.

Update performance is currently not comparable: Carrying out the MONDIAL data integration case study in XSLT would require a completely different program (according to the XSLT style which does not allow for incremental updates, but creates the result tree in a one-pass approach). As long as XML-QL and XQuery do not support updates, here also the tree must be generated in a single, large query. Due to their weak performance, the XML-QL and KWEELT systems are currently not suitable for such tasks.
Some details can be found in Section 15.6.

16.2 Contributions

The present work defines an XML data model suitable for data integration, and the logic-based language XPathLog which uses this data model for XML querying, manipulation, and integration. The data model and the language are implemented in the LOPiX system. Its practicability has been demonstrated by the MONDIAL case study.

Its principal difference to the XML Query Data Model and the DOM model is that it allows an element to have multiple parents. This data model supports the notion of an *XML database* which

[1] the problems of instability and documentation seem to be related ...

contains multiple, overlapping XML trees which represent (i) the sources, and (ii) result view(s) over the sources. The database is queried and manipulated by XPathLog. Special operations, i.e., *linking*, *element fusion*, definition of *synonyms*, and projection of result trees via an *export signature* provide special functionality needed for data integration. Additionally, an expressive extension with a class hierarchy for knowledge representation tasks has been defined.

XPathLog is completely XPath-based, i.e., both the rule bodies and the rule heads use an extended XPath syntax, thereby defining an update semantics for XPath expressions. The close relationship with XPath ensures that its declarative semantics is well understood from the XML perspective. Especially the nature of rule based bottom-up programming is easily understandable for XSLT practitioners, providing even more functionality. The Logic Programming background provides a strong theoretical foundation of the language concept.

Up to the publication of [TIHW01] (SIGMOD 2001), XPathLog was the first declarative, native XML language which allows for view definition and updates – at least, LoPiX [LoP] is the first system allowing for updates in a "native" XML language which was made publicly available. Still, XPathLog is the only approach which uses XPath syntax for *generating* or *updating* XML structure.

A Mondial DTD

The structure of the MONDIAL XML database is described by the following DTD:

```
<!-- XML DTD "mondial-2.0.dtd":
  (Wolfgang May, may@informatik.uni-freiburg.de, March 2000)
  a hierarchical DTD for the MONDIAL database,
  containing e.g.,
  - scalar reference attributes (city/capital)
  - multivalued reference attributes (organization/member/country)
  - cross-references in both directions (organization/member/country,
    country/memberships)
  - a "boolean"/flag attribute: city/is_country_cap
  - reference attributes with more than one target class
    (river/to, references rivers, lakes, and seas) -->

<!ELEMENT mondial (country*,continent*,organization*,
                  mountain*,(sea*,river*,lake*,desert*,island*)*)>
<!ELEMENT country (name,population,
                  population_growth?,infant_mortality?,
                  gdp_total?,gdp_agri?,gdp_ind?,gdp_serv?,
                  inflation?,indep_date?,government?,encompassed+,
                  ethnicgroups*,religions*,languages*,border*,
                  province*,city*)>
<!ATTLIST country car_code ID #IMPLIED
                 area CDATA #IMPLIED
                 capital IDREF #IMPLIED
                 memberships IDREFS #IMPLIED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT area (#PCDATA)>
<!ELEMENT population (#PCDATA)>
<!-- note that population is also a subelement of city -->
<!ATTLIST population year CDATA #IMPLIED>
<!ELEMENT population_growth (#PCDATA)>
<!ELEMENT infant_mortality (#PCDATA)>
<!ELEMENT gdp_total (#PCDATA)>
<!ELEMENT gdp_ind (#PCDATA)>
<!ELEMENT gdp_agri (#PCDATA)>
<!ELEMENT gdp_serv (#PCDATA)>
<!ELEMENT inflation (#PCDATA)>
<!ELEMENT indep_date (#PCDATA)>
<!ELEMENT government (#PCDATA)>
<!ELEMENT encompassed EMPTY>
<!ATTLIST encompassed continent IDREF #REQUIRED
                     percentage CDATA #REQUIRED>
<!ELEMENT ethnicgroups (#PCDATA)>
<!ATTLIST ethnicgroups percentage CDATA #REQUIRED>
```

```
<!ELEMENT religions (#PCDATA)>
<!ATTLIST religions percentage CDATA #REQUIRED>
<!ELEMENT languages (#PCDATA)>
<!ATTLIST languages percentage CDATA #REQUIRED>
<!ELEMENT border EMPTY>
<!ATTLIST border country IDREF #REQUIRED
                 length CDATA #REQUIRED
                 justice IDREF 'org-UN'>

<!ELEMENT province (name,area?,population,city*)>
<!ATTLIST province id ID #REQUIRED
                   country IDREF #REQUIRED
                   capital IDREF #IMPLIED>

<!ELEMENT city (name,longitude?,latitude?,
                population*,located_at*)>
<!ATTLIST city id ID #REQUIRED
               is_country_cap CDATA #IMPLIED
               is_state_cap CDATA #IMPLIED
               country IDREF #REQUIRED
               province IDREF #IMPLIED>
<!ELEMENT longitude (#PCDATA)>
<!ELEMENT latitude (#PCDATA)>
<!ELEMENT located_at EMPTY>
<!ATTLIST located_at watertype (river|sea|lake) #REQUIRED
                     river IDREFS #IMPLIED
                     sea IDREFS #IMPLIED
                     lake IDREFS #IMPLIED>

<!ELEMENT organization (name,abbrev,established?,members*)>
<!ATTLIST organization id ID #REQUIRED
                       seat IDREF #IMPLIED>
<!ELEMENT abbrev (#PCDATA)>
<!ELEMENT established (#PCDATA)>
<!ELEMENT members EMPTY>
<!ATTLIST members type CDATA #REQUIRED
                 country IDREFS #REQUIRED>

<!ELEMENT continent (name,area)>
<!ATTLIST continent id ID #REQUIRED>

<!ELEMENT river (length?,name,to*,located*)>
<!ATTLIST river id ID #REQUIRED
                country IDREFS #REQUIRED>

<!ELEMENT length (#PCDATA)>
<!ELEMENT to EMPTY>
<!ATTLIST to watertype (river|sea|lake) #REQUIRED
             water IDREF #REQUIRED>

<!ELEMENT located EMPTY>
<!ATTLIST located country IDREF #REQUIRED
                  province IDREFS #IMPLIED>
```

```
<!ELEMENT lake (name,area?,depth?,located*)>
<!ATTLIST lake id ID #REQUIRED
               country IDREFS #REQUIRED>
<!ELEMENT depth (#PCDATA)>

<!ELEMENT sea (name,area?,depth?,located*)>
<!ATTLIST sea id ID #REQUIRED
              country IDREFS #REQUIRED
              bordering IDREFS #IMPLIED>

<!ELEMENT desert (name,area?,located*)>
<!ATTLIST desert id ID #REQUIRED
                 country IDREFS #REQUIRED
                 climate NMTOKENS #FIXED 'dry aride'
                 temperature NMTOKEN 'hot'
                 ground (sand|boulders|rocks|snow) 'sand'>

<!ELEMENT island (name,area?,longitude?,latitude?,located*)>
<!ATTLIST island id ID #REQUIRED
                 country IDREFS #REQUIRED>

<!ELEMENT mountain (name,longitude?,latitude?,height?,located*)>
<!ATTLIST mountain id ID #REQUIRED
                   country IDREFS #REQUIRED>
<!ELEMENT height (#PCDATA)>
```

B Mondial XML SCHEMA

The XML Schema instance below describes the MONDIAL database.

```
<?xml version="1.0"?>
<schema xmlns          ="http://www.w3.org/1999/XMLSchema"

<complexType name="mondial">
   <element name="country"      type="country"      minOccurs="0" maxOccurs="unbounded"/>
   <element name="continent"    type="continent"    minOccurs="0" maxOccurs="unbounded"/>
   <element name="organization" type="organization" minOccurs="0" maxOccurs="unbounded"/>
   <element name="mountain"     type="mountain"     minOccurs="0" maxOccurs="unbounded"/>
   <element name="sea"          type="sea"          minOccurs="0" maxOccurs="unbounded"/>
   <element name="river"        type="river"        minOccurs="0" maxOccurs="unbounded"/>
   <element name="lake"         type="lake"         minOccurs="0" maxOccurs="unbounded"/>
   <element name="desert"       type="desert"       minOccurs="0" maxOccurs="unbounded"/>
   <element name="island"       type="island"       minOccurs="0" maxOccurs="unbounded"/>
</complexType>

<complexType name="country">
   <attribute name="car_code"    type="ID"       use="required"/>
   <attribute name="capital"     type="IDREF"    use="optional"/>
   <attribute name="memberships" type="IDREFS"   use="optional"/>
   <attribute name="industry"    type="NMTOKENS" use="optional"/>
   <element ref="name"/>
   <element ref="area"/>
   <element name="population"        type="integer" minOccurs="0" maxOccurs="1"/>
<complexType name="country">
   <attribute name="car_code"    type="ID"       use="required"/>
   <attribute name="capital"     type="IDREF"    use="optional"/>
   <attribute name="memberships" type="IDREFS"   use="optional"/>
   <attribute name="industry"    type="NMTOKENS" use="optional"/>
   <element ref="name"/>
   <element ref="area"/>
   <element name="population"        type="integer" minOccurs="0" maxOccurs="1"/>
   <element name="population_growth" type="decimal" minOccurs="0" maxOccurs="1"/>
   <element name="infant_mortality"  type="decimal" minOccurs="0" maxOccurs="1"/>
   <element name="gdp_total"         type="decimal" minOccurs="0" maxOccurs="1"/>
   <element name="gdp_agri"          type="decimal" minOccurs="0" maxOccurs="1"/>
   <element name="gdp_ind"           type="decimal" minOccurs="0" maxOccurs="1"/>
   <element name="gdp_serv"          type="decimal" minOccurs="0" maxOccurs="1"/>
   <element name="inflation"         type="decimal" minOccurs="0" maxOccurs="1"/>
   <element name="indep_date"        type="date"    minOccurs="0" maxOccurs="1"/>
   <element name="government"        type="string"  minOccurs="0" maxOccurs="1"/>
   <element name="ethnicgroup" type="culturalInfo"
            minOccurs="0"      maxOccurs="unbounded"/>
   <element name="religion"    type="culturalInfo"
```

```
            minOccurs="0"      maxOccurs="unbounded"/>
   <element name="language"    type="culturalInfo"
            minOccurs="0"      maxOccurs="unbounded"/>
   <element name="encompassed"  minOccurs="1" maxOccurs="unbounded"  >
     <complexType content="empty">
       <attribute name="continent"  type="IDREF"   use="required"/>
       <attribute name="percentage" type="decimal" use="required"/>
     </complexType>
   </element>
   <element name="border"  minOccurs="0" maxOccurs="unbounded"  >
     <complexType content ="empty">
       <attribute name="country"  type="IDREF"   use="required"/>
       <attribute name="length"   type="decimal" use="required"/>
     </complexType>
   </element>

   <element name="province"    type="province" minOccurs="0" maxOccurs="unbounded"/>
   <element name="city"        type="city"     minOccurs="0" maxOccurs="unbounded"/>
   <key name="countrykey">
      <selector>.</selector>
      <field>@car_code</field>
   </key>
   <keyref name="country2capital" refer="citykey">
      <selector>.</selector>
      <field>@capital</field>
   </keyref>
   <keyref name="encompassed2continent" refer="continentkey">
      <selector>encompassed</selector>
      <field>@continent</field>
   </keyref>
   <keyref name="border2country" refer="countrykey">
      <selector>border</selector>
      <field>@country</field>
   </keyref>
</complexType>

<complexType name="continent">
   <element ref="name"/>
   <element ref="area"/>
   <attribute name="id" type="ID" use="optional"/>
   <key name="continentkey">
      <selector>.</selector>
      <field>@id</field>
   </key>
</complexType>

<complexType name="organization">
   <attribute name="id"   type="ID"    use="required"/>
   <attribute name="seat" type="IDREF" use="required"/>
   <element ref="name"/>
   <element name="abbrev" type="string"/>
   <element name="established" type="date" minOccurs="0" maxOccurs="1"/>
   <element name="member" minOccurs="0" maxOccurs="unbounded">
      <complexType content="empty">
```

```
        <attribute name="type"    type="string" use="required"/>
        <attribute name="country" type="IDREFS"  use="required"/>
      </complexType>
   </element>
   <key name="organizationkey">
      <selector>.</selector>
      <field>@id</field>
   </key>
   <keyref name="org2seat" refer="citykey">
      <selector>.</selector>
      <field>@seat</field>
   </keyref>
</complexType>

<complexType name="mountain">
   <attribute name="id"      type="ID"      use="required"/>
   <attribute name="country" type="IDREFS"  use="required"/>
   <element ref="name"/>
   <element ref="longitude"  minOccurs="0"  maxOccurs="1"  />
   <element ref="latitude"   minOccurs="0"  maxOccurs="1"  />
   <element name="height"    type="integer" minOccurs="0" maxOccurs="1"  />
   <element ref="located"    minOccurs="0"  maxOccurs="unbounded"  />
</complexType>

<complexType name="sea">
   <attribute name="id"      type="ID"      use="required"/>
   <attribute name="country" type="IDREFS"  use="required"/>
   <element ref="name"/>
   <element ref="area"       minOccurs="0"  maxOccurs="1"  />
   <element name="depth"     type="integer" minOccurs="0" maxOccurs="1"/>
   <element ref="located"    minOccurs="0"  maxOccurs="unbounded"  />
</complexType>

<complexType name="river">
   <attribute name="id"      type="ID"      use="required"/>
   <attribute name="country" type="IDREFS"  use="required"/>
   <element name="length"    type="integer" minOccurs="0" maxOccurs="unbounded"/>
   <element ref="name"/>
   <element name="to"        minOccurs="0" maxOccurs="1">
      <complexType content="empty">
         <attribute name="type" type="water" use="required"/>
         <attribute name="ref"  type="IDREF"  use="required"/>
      </complexType>
   </element>
   <element ref="located"  minOccurs="0" maxOccurs="unbounded"/>
</complexType>


<complexType name="lake">
   <attribute name="id"      type="ID"      use="required"/>
   <attribute name="country" type="IDREFS"  use="required"/>
   <element ref="name"/>
   <element ref="area" minOccurs="0"    maxOccurs="1"/>
   <element name="depth"     type="integer" minOccurs="0" maxOccurs="1"/>
```

```
   <element ref="located"    minOccurs="0"  maxOccurs="unbounded"/>
</complexType>


<complexType name="desert">
   <attribute name="id"      type="ID"     use="required"/>
   <attribute name="country" type="IDREFS" use="required"/>
   <element ref="name"/>
   <element ref="area"       minOccurs="0" maxOccurs="1"/>
   <element ref="located"    minOccurs="0" maxOccurs="unbounded"/>
</complexType>

<complexType name="island">
   <attribute name="id"      type="ID"     use="required"/>
   <attribute name="country" type="IDREFS" use="required"/>
   <element ref="name"/>
   <element name="islands"   type="string" minOccurs="0" maxOccurs="1"/>
   <element ref="area" minOccurs="0" maxOccurs="1"/>
   <element ref="longitude"  minOccurs="0" maxOccurs="1"/>
   <element ref="latitude"   minOccurs="0" maxOccurs="1"/>
   <element ref="located"    minOccurs="0" maxOccurs="unbounded"/>
</complexType>

<!-- auxiliary declarations -->

<complexType name="culturalInfo" base="string" derivedBy="extension">
   <attribute name="percentage"  type="decimal"  use="required"/>
</complexType>

<element name="city"     type="city"/>
<element name="name"     type="string"/>
<element name="area"     type="integer"/>

<element name="longitude" type="decimal"/>
<element name="latitude" type="decimal"/>
<element name="located_at">
     <complexType content ="empty">
       <attribute name="type"  type="water"  use="required"/>
       <attribute name="river" type="IDREFS"  use="optional"/>
       <attribute name="sea"   type="IDREFS"  use="optional"/>
       <attribute name="lake"  type="IDREFS"  use="optional"/>
     </complexType>
</element>

<element name="located">
     <complexType content ="empty">
       <attribute name="country"  type="IDREF"   use="required"/>
       <attribute name="province" type="IDREFS"  use="optional"/>
     </complexType>
</element>

<simpleType name="water" base="string">
  <enumeration value="river"/>
  <enumeration value="sea"/>
```

```
  <enumeration value="lake"/>
</simpleType>

<complexType name="province">
    <element ref="name"/>
    <element ref="area"        minOccurs="0"  maxOccurs="1"/>
    <element name="population" type="integer" minOccurs="0" maxOccurs="1"/>
    <element name="city"       type="city"    minOccurs="0" maxOccurs="unbounded"/>
    <attribute name="id"       type="ID"      use="required"/>
    <attribute name="country"  type="IDREF"   use="optional"/>
    <attribute name="country"  type="IDREF"   use="optional"/>
    <attribute name="capital"  type="IDREF"   use="optional"/>
    <key name="provincekey">
      <selector>.</selector>
      <field>@id</field>
    </key>
    <keyref name="province2capital" refer="citykey">
      <selector>.</selector>
      <field>@capital</field>
    </keyref>
    <keyref name="province2country" refer="countrykey">
      <selector>.</selector>
      <field>@country</field>
    </keyref>
</complexType>

<complexType name="city">
    <attribute name="id"             type="ID"      use="required"/>
    <attribute name="country"        type="IDREF"   use="optional"/>
    <attribute name="province"       type="IDREF"   use="optional"/>
    <attribute name="is_country_cap" type="boolean" use="optional"/>
    <attribute name="is_state_cap"   type="boolean" use="optional"/>
    <element ref="name"/>
    <element ref="longitude"        minOccurs="0" maxOccurs="1"/>
    <element ref="latitude"         minOccurs="0" maxOccurs="1"/>
    <element name="population" minOccurs="0" maxOccurs="unbounded">
      <complexType base="integer" derivedBy="extension">
         <attribute name="year" type ="date" use="optional"/>
      </complexType>
    </element>
    <element ref="located_at"  minOccurs="0"  maxOccurs="unbounded"/>
    <key name="citykey">
      <selector>.</selector>
      <field>@id</field>
    </key>
    <keyref name="city2country" refer="countrykey">
      <selector>.</selector>
      <field>@country</field>
    </keyref>
    <keyref name="city2province" refer="provincekey">
      <selector>.</selector>
      <field>@province</field>
    </keyref>
</complexType>
```

```
</schema>
```

C MAPPING XML SCHEMA TO SIGNATURE ATOMS

The XML Schema metadata description can be used for deriving XPathLog signature atoms. The following XML Schema constructs are processed:

- *ComplexTypes*: they define classes which have properties given by their *element* and *attribute* children.
- *Elements, Attributes*: they define properties, either local when occurring as children of a *complexType*, or global to be used by children of a *complexType* via the ref attribute.
- maxOccurs and minOccurs define the cardinality of elements,
- use defines the cardinality of attributes,
- type defines the result class of a property,
- key and keyref can be used for deriving the result class or reference attributes.

```
mondial.xsd = "file:mondial-2.0.xsd" isa url.
U.parse@(xml)  :- U isa url.
schema = X :- U.parse@(xml) = B, B[schema->X].
?- sys.strat.doIt.
```

XML Schema predefined types are related to the XPathLog class hierarchy:

```
string subcl literal.
numeric subcl literal.
integer subcl numeric.
decimal subcl integer.
float subcl decimal.

nmtoken subcl string.
id subcl nmtoken.
nmtokens subcl string.
boolean subcl string.

simpletype subcl type.
complextype subcl type.

X isa simpletype :- X subcl literal.
date isa simpletype.

object[@idterm->object].
X[@idterm->X] :- X isa simpletype.

elementonly isa contentstype.
empty isa contentstype.
```

```
mixed isa contentstype.
textonly isa contentstype.

mvd isa cardinality.
sc  isa cardinality.
?- sys.strat.doIt.
```

Global vs. local declarations. Collect global element, attribute, and complexType definitions. Note that only attributes and elements which have a *name* attribute are local (excluding <element ref=" " ...> which is not a local declaration but a reference to a declaration):

```
X[@global->1] :- schema[M->X].
X[@uses_local->Y], Y[@local->1] :- X[complextype->Y], not X = schema.
X[@uses_local->Y], Y[@local->1] :- X[element->Y[@name->N]], not X = schema.
X[@uses_local->Y], Y[@local->1] :- X[attribute->Y[@name->N]], not X = schema.
```

Detect local declarations and propagate the *use* relation downwards (except if a "more local" thing with the same name is defined):

```
C[@uses_local->X] :- N[complextype->C], N[@uses_local->X isa Type[@name->V]],
                     not N[@uses_local->_ isa Type[@name->V]].
C[@uses_local->X] :- N[element->C], N[@uses_local->X isa Type[@name->V]],
                     not N[@uses_local->_ isa Type[@name->V]].
C[@uses_local->X] :- N[attribute->C], N[@uses_local->X isa Type[@name->V]],
                     not N[@uses_local->_ isa Type[@name->V]].
?- sys.strat.doIt.
```

Associate id-terms or references with types: global types are associated with their names as mnemonic constants, e.g.,

<complexType name="country">

is associated with the id-term country. Local types are associated with the reference *type.name* where *type* is the type which defines them. They become a subclass of *name*.

Note that this leads to a "dirty" class hierarchy if a global type is "redefined" locally with a local type which does not define a subtype.

```
X[@idterm->I] :- X isa element[@name->EN], string2object(EN,I).
X[@idterm->I] :- X isa attribute[@name->AN], string2object(AN,I).
X[@idterm->I subcl object] :- X isa type[@global->1 and @name->N], string2object(N,I).
```

ComplexTypes which are defined local to type declarations (possibly inside local element declarations):

```
X[@idterm->(T:I) subcl object], T:I subcl I :-
            _T isa type[@idterm->T and @uses_local->X isa type[@name->N]],
            string2object(N,I).
X[@idterm->(T:I) subcl object], T:I subcl I :-
            _T isa type[@idterm->T]/element[@idterm->I and @uses_local->X isa type].
X[@idterm->(T:I) subcl object], T:I subcl I :-
            _T isa type[@idterm->T]/attribute[@idterm->I and @uses_local->X isa type].
```

ComplexTypes which are defined local to global element declarations (i.e., which are not local to any type):

```
X[@idterm->T subcl object] :-
            _E isa element[@idterm->T and @uses_local->X isa type], not X/@name,
            not _ isa type[@uses_local->X],
            not _ isa type/element[@uses_local->X].
?- sys.strat.doIt.
```

Element and Attribute Declarations.
Element and attribute declarations have the following properties:

element[@name⇒string and @resulttype ⇒type and
@card⇒cardinality and @mincard⇒integer and
@minoccurs⇒string and @maxoccurs⇒string].

For deriving the result types, the use of local definitions has to be considered.
First, declarations which define a local type by themselves are handled, i.e.,

<element/attribute name="..."> <complex/simpletype .../> </element>

```
X[@resulttype->T] :- X isa element[complextype->T].
X[@resulttype->T] :- X isa element[simpletype->T].
X[@resulttype->T] :- X isa attribute[simpletype->T].
```

Some declarations use a local type defined by one of their ancestors by

<element/attribute type="..." ...>

```
X[@resulttype->T] :- X isa element[@type->TN],
                           X[@uses_local->T isa type[@name->TN]].
?- sys.strat.doIt.
```

All others use global declarations via the same form

<element/attribute type="..." ...>

- using defined types:

```
X[@resulttype->T] :- X isa element[@type->TN],
                           not X[@uses_local->_ isa type[@name->TN]],
                           T[@global->1 and @name->TN].
X[@resulttype->T] :- X isa attribute[@type->TN],
                           not X[@uses_local->_ isa type[@name->TN]],
                           T[@global->1 and @name->TN].
```

- using predefined simple types:

```
X[@resulttype->T] :- X[@type->TN],
                           not X[@uses_local->_ isa type[@name->TN]],
                           string2object(TN,T), T isa simpletype.
```

- for attributes which are of type IDREF or IDREFS, we do only know that the result is an object:

```
X[@resulttype->object] :- X isa attribute[@type->"IDREF"].
X[@resulttype->object] :- X isa attribute[@type->"IDREFS"].

?- sys.strat.doIt.
```

- If the given result type of a subelement is a literal type, it is in fact an element with PCDATA contents. Create a new type with text contents as an “annotated literal type”. The new type is a subclass of the literal type:

```
E[@resulttype->E], E subcl ResType :-
       E isa element[@global->1 and @idterm->I and
                     @resulttype->ResType subcl literal].

E[@resulttype->(T:I) subcl I[@idterm->(T:I) and text()=>ResType]],
I subcl object, (T:I) subcl ResType :-
       E isa element[@local->1 and @idterm->I and @resulttype->ResType],
       _ isa type[@idterm->T and @uses_local->E],
       ResType subcl literal, not ResType subcl object.
```

- Analogously, complexTypes whose base type is a literal type (i) are a subclass (a refinement) of the base type and (ii) provide a text method for accessing the element contents which is actually a literal value of the given base type:

```
I subcl BT, I[text()=>BT] :-
   T isa complextype[@idterm->I and @base->B],
   string2object(B,BT), BT subcl literal.
?- sys.strat.doIt.
```

Element and attribute definitions can also refer to other declarations.

<element/attribute ref=”...”>

Again, it must be checked if a global or a local declaration is referred to:

```
X[@name->EN and @idterm->I and @resulttype->T] :-
     X isa element[@ref->EN and
                   @uses_local->_ isa element[@idterm->I and @name->EN and
                                              @resulttype->T]].
X[@name->EN and @idterm->I and @resulttype->T] :-
     X isa element[@ref->EN], not X[@uses_local->_ isa element[@name->EN]],
     E isa element[@global->1 and @name->EN and @idterm->I and @resulttype->T].
X[@name->AN and @idterm->I and @resulttype->T] :-
     X isa attribute[@ref->AN and
                     @uses_local->_ isa attribute[@name->AN and @idterm->I and
                                                  @resulttype->T]].
X[@name->AN and @idterm->I and @resulttype->T] :-
     X isa attribute[@ref->AN], not X[@uses_local->_ isa attribute[@name->AN]],
     A isa attribute[@global->1 and @name->AN and @idterm->I and @resulttype->T].
?- sys.strat.doIt.
```

Now, all <element> and <attribute> elements have a name (a string) and all <element> elements have a xsd.resulttype (reference) property. Attributes do not yet have a result type.

```
?- sys.strat.doIt.
```

The cardinality of subelements is given by the @minoccurs and @maxoccurs attributes of <element> elements:

```
element[@minoccurs*->"1"].
X[@maxoccurs->MIN] :- X isa element[@minoccurs->MIN],
                      not X isa element[@maxoccurs->_], not MIN = "0".
?- sys.strat.doIt.
element[@maxoccurs*->"1"].
?- sys.strat.doIt.
```

Attribute cardinality is given by their use attribute: Note that IDREFS and NMTOKENS attributes are multivalued in our approach.

```
X[@minoccurs->"1"] :-  X isa attribute[@use->"required"].
X[@minoccurs->"0"] :-  X isa attribute [@use->"optional"].
X[@minoccurs->"1"] :-  X isa attribute [@use->"default"].
X[@minoccurs->"1"] :-  X isa attribute [@use->"fixed"].
X[@minoccurs->"0"] :-  X isa attribute [@use->"prohibited"].
attribute[@minoccurs*->"0"].
?- sys.strat.doIt.
X[@maxoccurs->"0"] :-  X isa attribute [@use->"prohibited"].
X[@maxoccurs->"unbounded"] :-  X isa attribute[@type->"NMTOKENS"],
                               not X[@use->"prohibited"].
X isa attribute[@maxoccurs->"unbounded"] :-  X isa attribute[@type->"IDREFS"],
                               not  X isa attribute [@use->"prohibited"].
X isa attribute[@maxoccurs->"1"] :-  X isa attribute[@use->"required"],
                                 not X isa attribute[@type->"NMTOKENS"],
                                 not X isa attribute[@type->"IDREFS"].
X isa attribute[@maxoccurs->"1"] :-  X isa attribute[@use->"optional"],
                                 not X isa attribute[@type->"NMTOKENS"],
                                 not X isa attribute[@type->"IDREFS"].
X isa attribute[@maxoccurs->"1"] :-  X isa attribute[@use->"default"],
                                 not X isa attribute[@type->"NMTOKENS"],
                                 not X isa attribute[@type->"IDREFS"].
X isa attribute[@maxoccurs->"1"] :-  X isa attribute[@use->"fixed"],
                                 not X isa attribute[@type->"NMTOKENS"],
                                 not X isa attribute[@type->"IDREFS"].
?- sys.strat.doIt.
attribute[@maxoccurs*->"1"].
?- sys.strat.doIt.
```

The @mincard and card $\in \{scalar, multivalued\}$ properties are derived:

```
X[@mincard->MIN] :- X[@minoccurs->MIN].
X[@card->mv] :- X[@maxoccurs->M], string2integer(M,I), I > 1.
X[@card->sc] :- X[@maxoccurs->M], string2integer(M,I), I = 1.
X[@card->sc] :- X[@maxoccurs->M], string2integer(M,I), I = 0.
X[@card->mv] :- X[@maxoccurs->M], M = "unbounded".
?- sys.strat.doIt.
```

Now, all attributes and subelement properties are defined with name, cardinality, and result type.

Content Models.
First it is derived which types have empty, text-only, element-only, or mixed contents:

```
type[@contents=>contentstype and attribute=>attribute and element=>element].
```

If a type has an explicit contents specification, take it:

```
X[@contents->CSPEC] :- X isa type[@content->CSPEC].
```

If a there is no content attribute take the specification from the base type:

```
X[@contents->CSPEC] :-
       X isa type[not @content and @base/@content->CSPEC].
?- sys.strat.doIt.
```

Enumerations. Enumeration types define classes. All enumerated values are members of that class.

```
enumerated[@name=>string and @values=>object].
ENUM isa enumerated[@idterm->CL and @name->NA and @values->VAL isa CL] :-
         X[@resulttype->ENUM[@name->NA and enumeration->ITEM]],
         ITEM[@value->V], string2object(NA,CL),string2object(V,VAL).
?- sys.strat.doIt.
```

Signatures. Associate signatures with types. Global types define classes by their *name* attribute. Local types are local wrt. their environment type.

Take only the non-literal result types (recall that non-literal result types have already been replaced by "annotated literals" having text-only contents):

```
Type[M=>Res] :-
       _ isa complextype[@idterm->Type and
             element->_[@idterm->M and @resulttype->ResType[@idterm->Res]]].
?- sys.strat.doIt.
```

Non-reference attributes have only literal result types, reference attributes have object results:

```
Type[@M=>Res] :-
       _ isa complextype[@idterm->Type and
             attribute->_[@idterm->M and @resulttype->ResType[@idterm->Res]]].
?- sys.strat.doIt.
?-sys.echo@("keys").
```

Keys. Keys are used as targets for reference attributes. Thus, they can serve for deriving the signature of reference attributes (if these are described by keyref elements). First, for every key, it is detected for which class it is a key:

```
K[@identifies->RS] :-
  T isa type[@idterm->TI and key->K isa key[selector/text()->Sel]],
  not Sel=".", string2object(Sel,SID),
  TI[SID=>RS], not RS subcl literal.
K[@identifies->RS] :-
  T isa type[@idterm->TI and key->K isa key[selector/text()->Sel]],
  not Sel=".", string2object(Sel,SID),
  TI[SID=>RS], not RS subcl literal.
K[@identifies->TI] :-
  T isa type[@idterm->TI and key->K isa key[selector/text()->Sel]],
  Sel=".".
?- sys.strat.doIt.
```

For every keyref element, it is detected which is the host class for the attribute it describes:

```
KR[@hostclass->TI] :-
  T isa type[@idterm->TI and keyref->KR isa keyref[selector/text()->Sel]],
  Sel=".".
KR[@hostclass->X] :-
  T isa type[@idterm->TI and keyref->KR isa keyref[selector/text()->Sel]],
  not Sel=".",
  T[@idterm->TI and element->E[@name->Sel and @idterm->MI]],
  TI[MI=>X].
?- sys.strat.doIt.
```

Then, the method which is described by the key reference is derived:

```
KR[@method->MI] :-
  KR isa keyref[@hostclass->TI and field/text()->F],
  pmatch(F,"/\A@(.*)\Z/",["$1"],[Attr]),
  HC[@idterm->TI and attribute->A[@name->Attr and @idterm->MI]].
?- sys.strat.doIt.
```

Finally, the key used by the foreign key reference is used for associating the result class (the one identified by the key) with the property (which is described by the keyref element):

```
TI[MI=>RS] :-
  KR isa keyref[@hostclass->TI and @method->MI and @refer->KN],
  TI[MI=>object],
  K isa key[@name->KN and @identifies->RS].
?- sys.strat.doIt.
?- sys.theOMAccess.export@("sig","mondial-2.0-sig.lpx").
```

Mondial Signature

The XPathLog signature given below is extracted from the XML Schema specification. Note that the local type declarations are namespaced.

```
mondial :: object.
mondial [country => (country);
  continent => (continent);
  organization => (organization);
  mountain => (mountain);
  sea => (sea);
  river => (river);
  lake => (lake);
  desert => (desert);
  island => (island)].

country :: object.
country [name => (string, name);
  area => (integer, area);
  population => (integer,
    country:population);
```

```
  population_growth => (decimal,
    country:population_growth);
  infant_mortality => (decimal,
    country:infant_mortality);
  gdp_total => (decimal,
    country:gdp_total);
  gdp_agri => (decimal,
    country:gdp_agri);
  gdp_ind => (decimal,
    country:gdp_ind);
  gdp_serv => (decimal,
    country:gdp_serv);
  inflation => (decimal,
    country:inflation);
  indep_date => (date);
  government => (string,
    country:government);
  ethnicgroup => (culturalinfo);
  religion => (culturalinfo);
  language => (culturalinfo);
  encompassed => (country:encompassed);
  border => (country:border);
  province => (province);
  city => (city);
  @car_code => (id);
  @capital => (object);
  @memberships => (object);
  @industry => (nmtokens)].

continent :: object.
continent [name => (string, name);
  area => (integer, area);
  @id => (id)].

organization :: object.
organization [name => (string, name);
  abbrev => (string, organization.abbrev);
  established => (date);
  member => (organization.member);
  @id => (id);
  @seat => (object)].

mountain :: object.
mountain [name => (string, name);
  longitude => (decimal, longitude);
  latitude => (decimal, latitude);
  height => (integer, mountain.height);
  located => (located);
  @id => (id);
  @country => (object)].

sea :: object.
sea :: water.
sea [name => (string, name);
```

```
  area => (integer, area);
  located => (located);
  depth => (integer, sea.depth);
  @id => (id);
  @country => (object)].

river :: object.
river :: water.
river [length => (integer, river.length);
  name => (string, name);
  located => (located);
  to => (river.to);
  @id => (id);
  @country => (object)].

lake :: object.
lake :: water.
lake [name => (string, name);
  area => (integer, area);
  located => (located);
  depth => (integer, lake.depth);
  @id => (id);
  @country => (object)].

desert :: object.
desert [name => (string, name);
  area => (integer, area);
  located => (located);
  @id => (id);
  @country => (object)].

island :: object.
island [name => (string, name);
  area => (integer, area);
  longitude => (decimal, longitude);
  latitude => (decimal, latitude);
  located => (located);
  islands => (string, island.islands);
  @id => (id);
  @country => (object)].

province :: object.
province [name => (string, name);
  area => (integer, area);
  population => (integer, province.population);
  city => (city);
  @id => (id);
  @country => (object);
  @capital => (object)].

city :: object.
city [name => (string, name);
  population => (city.population);
  longitude => (decimal, longitude);
```

```
  latitude => (decimal, latitude);
  located_at => (located_at);
  @id => (id);
  @country => (object);
  @province => (object);
  @is_country_cap => (boolean);
  @is_state_cap => (boolean)].

culturalinfo :: string.
culturalinfo :: object.
culturalinfo [text() => (string);
  @percentage => (decimal)].

city.population :: object.
city.population :: integer.
city.population [text() => (integer);
  @year => (date)].

country:government :: string.
country:government :: object.
country:government [text() => (string)].

organization.abbrev :: string.
organization.abbrev :: object.
organization.abbrev [text() => (string)].

island.islands :: string.
island.islands :: object.
island.islands [text() => (string)].

country:population :: object.
country:population :: integer.
country:population [text() => (integer)].

mountain.height :: object.
mountain.height :: integer.
mountain.height [text() => (integer)].

sea.depth :: object.
sea.depth :: integer.
sea.depth [text() => (integer)].

river.length :: object.
river.length :: integer.
river.length [text() => (integer)].

lake.depth :: object.
lake.depth :: integer.
lake.depth [text() => (integer)].

province.population :: object.
province.population :: integer.
province.population [text() => (integer)].
```

```
country:population_growth :: object.
country:population_growth :: decimal.
country:population_growth [text() => (decimal)].

country:infant_mortality :: object.
country:infant_mortality :: decimal.
country:infant_mortality [text() => (decimal)].

country:gdp_total :: literal.
country:gdp_total :: integer.
country:gdp_total [text() => (decimal)].

country:gdp_agri :: object.
country:gdp_agri :: integer.
country:gdp_agri [text() => (decimal)].

country:gdp_ind :: object.
country:gdp_ind :: integer.
country:gdp_ind [text() => (decimal)].

country:gdp_serv :: object.
country:gdp_serv :: integer.
country:gdp_serv [text() => (decimal)].

country:inflation :: object.
country:inflation :: decimal.
country:inflation [text() => (decimal)].
```

D DTDS OF Mondial XML SOURCES

The original HTML MONDIAL data sources (described in the introduction) have been wrapped in [May99a] and exported into XML (available at [May01a]). The DTDs are given below.

D.1 CIA World Factbook Country Listing

```
<!-- XML DTD "cia-export.dtd":
  (Wolfgang May, may@informatik.uni-freiburg.de, Oct 2000) -->


<!ELEMENT cia (continent*, country*)>

<!ELEMENT continent EMPTY>
<!ATTLIST continent id ID #REQUIRED
                    name CDATA #REQUIRED>

<!ELEMENT name (#PCDATA)>

<!ELEMENT country (ethnicgroups*, religions*, languages*, borders*, coasts*)>

<!ATTLIST country
  id ID #REQUIRED
  name CDATA #REQUIRED
  datacode CDATA #IMPLIED
  continent CDATA #IMPLIED
  total_area CDATA #IMPLIED
  population CDATA #IMPLIED
  population_growth CDATA #IMPLIED
  infant_mortality CDATA #IMPLIED
  gdp_agri CDATA #IMPLIED
  gdp_ind CDATA #IMPLIED
  gdp_serv CDATA #IMPLIED
  gdp_total CDATA #IMPLIED
  inflation CDATA #IMPLIED
  gdp_total CDATA #IMPLIED
  indep_date CDATA #IMPLIED
  government CDATA #IMPLIED
  capital CDATA #IMPLIED>

<!ELEMENT ethnicgroups (#PCDATA)>
<!ATTLIST ethnicgroups name CDATA #REQUIRED>

<!ELEMENT religions (#PCDATA)>
<!ATTLIST religions name CDATA #REQUIRED>
```

```
<!ELEMENT languages (#PCDATA)>
<!ATTLIST languages name CDATA #REQUIRED>

<!ELEMENT borders (#PCDATA)>
<!ATTLIST borders country IDREF #REQUIRED>

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

D.2 CIA World Factbook Organizations

```
<!-- XML DTD "orgs-export.dtd":
  (Wolfgang May, may@informatik.uni-freiburg.de, Oct 2000) -->

<!ELEMENT orgs (organization*)>

<!ELEMENT organization (member_names*)>
<!ATTLIST organization id ID #REQUIRED
                       abbrev CDATA #REQUIRED
                       name CDATA #REQUIRED
                       established CDATA #IMPLIED
                       seatcity CDATA #IMPLIED
                       seatcountry CDATA #IMPLIED>

<!ELEMENT member_names (#PCDATA)>
<!ATTLIST member_names type CDATA #REQUIRED>
```

D.3 Global Statistics

```
<!-- XML DTD "gs-export.dtd":
  (Wolfgang May, may@informatik.uni-freiburg.de, Oct 2000) -->

<!ELEMENT gs (continent*, country*, city*, province*)>

<!ELEMENT continent EMPTY>
<!ATTLIST continent id ID #REQUIRED
                    name CDATA #REQUIRED>

<!ELEMENT name (#PCDATA)>

<!ELEMENT country (name+)>
<!ATTLIST country id ID #REQUIRED
                  capital IDREF #IMPLIED
                  population CDATA #IMPLIED
                  continent IDREF #IMPLIED
                  main_cities IDREFS #IMPLIED
                  adm_divs IDREFS #IMPLIED>


<!ELEMENT province EMPTY>
<!ATTLIST province id ID #REQUIRED
```

```
                  name CDATA #REQUIRED
                  country IDREF #REQUIRED
                  capital IDREF #IMPLIED
                  population CDATA #IMPLIED
                  area CDATA #IMPLIED>

<!ELEMENT city (population*, name+)>
<!ATTLIST city id ID #REQUIRED
               country IDREF #REQUIRED
               province IDREF #IMPLIED>

<!ELEMENT population (#PCDATA)>
<!ATTLIST population year CDATA #REQUIRED>
```

D.4 Qiblih Coordinates

```
<!-- XML DTD "qiblih-export.dtd":
  (Wolfgang May, may@informatik.uni-freiburg.de, Oct 2000) -->

<!ELEMENT qiblih (city*)>

<!ELEMENT city EMPTY>
<!ATTLIST city id ID #REQUIRED
               name CDATA #REQUIRED
               country CDATA #REQUIRED
               longitude CDATA #REQUIRED
               latitude CDATA #REQUIRED
               province CDATA #IMPLIED>
```

D.5 Terra

```
<!-- XML DTD "terra-export.dtd":
  (Wolfgang May, may@informatik.uni-freiburg.de, Oct 2000) -->

<!ELEMENT terra (country*, province*, city*,
   (mountain, desert, island, river, lake, sea)*)>

<!ELEMENT country (encompassed*)>
<!ATTLIST country id ID #REQUIRED
                  name CDATA #REQUIRED
                  code CDATA #REQUIRED
                  area CDATA #REQUIRED
                  population CDATA #REQUIRED
                  capital CDATA #REQUIRED>

<!ELEMENT encompassed (#PCDATA)>
<!ATTLIST encompassed continent CDATA #REQUIRED>

<!ELEMENT province (#PCDATA)>
<!ATTLIST province id ID #IMPLIED
                   name CDATA #IMPLIED
```

```
                   abbrev CDATA #IMPLIED
                   country CDATA #IMPLIED
                   pop CDATA #IMPLIED
                   capital CDATA #IMPLIED>

<!ELEMENT city (province*)>
<!ATTLIST city id ID #REQUIRED
               name CDATA #REQUIRED
               country CDATA #REQUIRED
               population CDATA #IMPLIED
               longitude CDATA #IMPLIED
               latitude CDATA #IMPLIED>

<!ELEMENT river (to*, located*)>
<!ATTLIST river id ID #REQUIRED
                name CDATA #REQUIRED
                length CDATA #IMPLIED>

<!ELEMENT to EMPTY>
<!ATTLIST to type (river|sea|lake) #REQUIRED
             water IDREF #REQUIRED>

<!ELEMENT located EMPTY>
<!ATTLIST located country_code CDATA #REQUIRED
                  province_id CDATA #REQUIRED>

<!ELEMENT lake (located*)>
<!ATTLIST lake id ID #REQUIRED
               name CDATA #REQUIRED
               area CDATA #IMPLIED>

<!ELEMENT sea (located*)>
<!ATTLIST sea id ID #REQUIRED
              name CDATA #REQUIRED
              depth CDATA #IMPLIED
              bordering IDREFS #REQUIRED>

<!ELEMENT desert (located*)>
<!ATTLIST desert id ID #REQUIRED
                 name CDATA #REQUIRED
                 area CDATA #IMPLIED>

<!ELEMENT island (located*)>
<!ATTLIST island id ID #REQUIRED
                 name CDATA #REQUIRED
                 area CDATA #IMPLIED
                 longitude CDATA #IMPLIED
                 latitude CDATA #IMPLIED>

<!ELEMENT mountain (located*)>
<!ATTLIST mountain id ID #REQUIRED
                   name CDATA #REQUIRED
                   height CDATA #REQUIRED
                   longitude CDATA #IMPLIED
```

```
                   latitude CDATA #IMPLIED>
```

D.6 Country Names and Codes

```
<!-- XML DTD "codes-export.dtd":
  (Wolfgang May, may@informatik.uni-freiburg.de, Oct 2000) -->

<!ELEMENT codes (country*)>

<!ELEMENT country (name,name,name)>
<!ATTLIST country id CDATA #REQUIRED
                  car_code CDATA #REQUIRED>

<!ELEMENT name (#PCDATA)>
<!ATTLIST name language CDATA #REQUIRED>
```

E LISTS

List of Theorems.

List of Definitions.

List of Examples.

List of Figures

Bibliography

For every paper, the page numbers where it is cited are given in parentheses.

[Abi97] S. Abiteboul. Querying Semi-Structured Data. In *Intl. Conference on Database Theory (ICDT)*, LNCS, no. 1186, pp. 1–18. Springer, 1997. (p 5)

[Ade98] B. Adelberg. NoDoSE – A Tool for Semi-Automatically Extracting Semi-Structured Data from Text Documents. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 283–294, 1998. (p 207)

[AM98] G. O. Arocena and A. O. Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. In *Intl. Conference on Data Engineering (ICDE)*, 1998. (p 205)

[AMM97] P. Atzeni, G. Mecca, and P. Merialdo. To Weave the Web. In *Intl. Conf. on Very Large Data Bases (VLDB)*, 1997. (p 209)

[AQM+97a] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Intl. Journal on Digital Libraries (JODL)*, 1(1), 1997. (pp 18, 38, 63, 67, 121)

[AQM+97b] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Intl. Journal on Digital Libraries (JODL)*, 1(1):68–88, 1997. (pp 5, 208)

[ASL89] A. Alashqur, S. Su, and H. Lam. OQL: A query Language for Manipulating Object-Oriented Databases. In *Intl. Conf. on Very Large Data Bases (VLDB)*, pp. 433–442, 1989. (p 55)

[AV97] S. Abiteboul and V. Vianu. Correspondence and Translation for Heterogeneous Data. In *Intl. Conference on Database Theory (ICDT)*, LNCS, no. 1186, pp. 351–363. Springer, 1997. (p 208)

[BDHS96] P. Buneman, S. Davidson, G. Hillebrandt, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 505–516, Montreal, Canada, 1996. (pp 5, 207, 208)

[Beh01] E. Behrends. Realisierung einer DOM-Speicherung für XPathLog. Diplomarbeit, Universität Freiburg (in german), 2001. (pp 128, 143, 197)

[BGL+99] C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-Based Information Mediation with MIX. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1999. (pp 18, 63, 67, 210)

[Bun97] P. Buneman. Semistructured Data (invited tutorial). In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 117–121, Tucson, Arizona, 1997. (pp 5, 37, 75)

[CB00] R. Cattell and D. Barry. *The Object Database Standard: ODMG 3.0.* Morgan Kaufmann, 2000. (pp 39, 148)

[CCD+99] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a Graphical Language for Querying and Restructuring XML Documents. In *Proc. 8th International World Wide Web Conference (WWW 8)*, 1999. (p 209)

[CCS00] V. Christophides, S. Cluet, and J. Siméon. On Wrapping Query Languages and Efficient XML Integration. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 141–152, 2000. (pp 18, 67, 209)

[CDSS99] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators need Data Conversion. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1999. (pp 18, 67, 121, 209)

[CFI+00] M. J. Carey, D. FLorescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPeranto: Publishing object-relational data as XML. In *WebDB 2000*, 2000. (p 211)

[CG01] L. Cardelli and G. Ghelli. A Query Language Based on the Ambient Logic. In *Europ. Symp. on Programming (ESOP); part of ETAPS 2001*, LNCS, no. 2028, pp. 1–22. Springer, 2001. (p 209)

[CHS+95] M. Carey, L. Haas, P. Schwarz, M. Arya, W.Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkuvic, J. Thomas, J. Williams, and E. Wimmers. Towards Heterogeneous Multimedia Information Systems. In *Research Issues in Data Engineering (RIDE)*, 1995. (p 209)

[Cla] J. Clark. SP: An SGML System Conforming to International Standard ISO 8879 – Standard Generalized Markup Language. `http://www.jclark.com/sp`. (p 195)

[Cla98] J. Clark. XT: an implementation of XSL Transformations. `http://www.jclark.com/xml/xt.html`, 1998. (pp 34, 200)

[CRF00] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *WebDB 2000*, pp. 53–62, 2000. (pp 1, 18, 26, 54, 55, 72)

[DFF+98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. `http://www.w3.org/TR/1998/NOTE-xml-ql`, 1998. (pp 17, 34, 208)

[DFF+99a] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu. Querying XML Data. *IEEE DATA Engineering Bulletin*, 22(3), 1999. (pp 17, 34)

[DFF+99b] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. In *8th. WWW Conference*. W3C, 1999. World Wide Web Consortium Technical Report, NOTE-xml-ql-19980819, `www.w3.org/TR/NOTE-xml-ql`. (pp 1, 17, 26, 34, 37, 75, 122, 123, 208)

[DFF+99c] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. `http://www.research.att.com/sw/tools/xmlql`, 1999. (pp 18, 38, 63, 201)

[DFS00] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 431–442, 2000. (pp 110, 211)

[DOM98] Document Object Model (DOM). `http://www.w3.org/DOM/`, 1998. (pp 10, 75, 96)

[ECY+98] D. Embley, D. Campbell, Y. Yiang, Y.-K. Ng, and R. Smith. A Conceptual-Modeling Approach to Extracting Data from the Web. In *Intl. Conf. on the Entity Relationship Approach (ER)*, LNCS, no. 1507. Springer, 1998. (p 207)

[eXc] eXcelon Corp. XML Application Development using eXcelon. `http://www.exceloncorp.com/`. (pp 7, 17, 19, 30, 34, 59, 60, 62, 63, 65, 125, 201, 211) [eXc00] eXcelon Corporation. Vorteile persistenter Datenhaltung von XML in DOM-Repräsentation. In *Invited Talk, Workshop Internet-Datenbanken, GI-Jahrestagung*, 2000. (p 19)

[FFK+98] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the Boat with Strudel: Experiences with a Web-Site Management System. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1998. (pp 17, 34, 37, 38, 63, 66, 67, 75, 127, 207, 208)

[FFLS97] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for a Web-Site Management System. *SIGMOD Record*, 26(3):4–11, 1997. (pp 17, 34, 207, 208)

[FHK+97] J. Frohn, R. Himmeröder, P.-T. Kandzia, G. Lausen, and C. Schlepphorst. FLORID: A Prototype for F–Logic. In *Intl. Conf. on Data Engineering (ICDE)*, 1997. (p 136)

[FK99] D. Florescu and D. Kossman. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report 3684, INRIA, 1999. (pp 110, 211)

[FLM98] D. Florescu, A. Levy, and A. Mendelzon. Database Techniques for the World-Wide Web: A Survey. *SIGMOD Record*, 27(3), September 1998. (p 209)

[FLO98] FLORID Homepage. `http://www.informatik.uni-freiburg.de/~dbis/florid/`, 1998. (pp 3, 136, 193)

[FLU94] J. Frohn, G. Lausen, and H. Uphoff. Access to Objects by Path Expressions and Rules. In *Intl. Conf. on Very Large Data Bases (VLDB)*, pp. 273–284, 1994. (pp 115, 188)

[Fro98] J. Frohn. *Magic-Set Transformation in deduktiven, objektorientierten Datenbanksprachen*. PhD thesis, Institut für Informatik, Universität Freiburg, 1998. (p 120)

[FSW99] M. Fernandez, J. Siméon, and P. Wadler. XML Query Languages: Experiences and Exemplars. draft manuscript, communication to the XML Query W3C Working Group, 1999. `http://www-db.research.bell-labs.com/user/simeon/xquery.ps`. (pp 18, 121)

[FTS00] M. Fernandez, W.-C. Tan, and D. Suciu. Silk Route: Trading between Relations and XML. In *Proc. 9th International World Wide Web Conference (WWW 9)*, 2000. (pp 18, 63, 210, 211)

[GHR94] D. M. Gabbay, C. J. Hogger, and J. A. Robinson. *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford Science Publications, 1994. (pp 250, 252)

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2), 1997. (pp 38, 63, 66, 67, 207)

[GMW99] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *WebDB 1999*, 1999. (pp 18, 38, 121, 208, 210)

[HC96] M. Hermenegildo and D. Cabeza. Internet and WWW Programming using Computational Logic Systems [PiLLoW]. In *Workshop on Logic Programming and the Internet*, 1996. `www.clip.dia.fi.upm.es/miscdocs/pillow/pillow.html`. (p 207)

[HFAN98] G. Huck, P. Fankhauser, K. Aberer, and E. J. Neuhold. Jedi: Extracting and Synthesizing Information from the Web. In *Intl. Conference on Cooperative Information Systems (CoopIS)*, pp. 32–43. IEEE Computer Science Press, 1998. (p 206)

[Him94] R. Himmeröder. Eine Objekt-Algebra zur Auswertung von Frame-Logik-Programmen. Master's thesis, Universität Mannheim, 1994. (pp 189, 191)

[HKL+98] R. Himmeröder, P.-T. Kandzia, B. Ludäscher, W. May, and G. Lausen. Search, Analysis, and Integration of Web Documents: A Case Study with FLORID. In *Proc. Intl. Workshop on Deductive Databases and Logic Programming (DDLP'98)*, pp. 47–57, 1998. (pp 63, 64, 66, 67)

[HM99] G. Huck and I. Macherius. GMD IPSI XQL Engine. `http://xml.darmstadt.gmd.de/xql/`, 1999. (pp 17, 30)

[Hor94] J. F. Horty. Some direct Theories of Nonmonotonic Inheritance. In *Handbook of Logic in Artificial Intelligence and Logic Programming* [GHR94]. (p 136)

[HP00] H. Hosoya and B. C. Pierce. XDuce: A Typed XML Processing Language. In *WebDB 2000*, 2000. (p 209)

[Kay99] M. Kay. SAXON: an XSLT processor. `http://www.users.iclway.co.uk/mhkay/saxon`, 1999. (p 34)

[KIF98] Knowledge Interchange Format (KIF); also ANSI Draft NCITS.T2/98-004. `http://logic.stanford.edu/kif/kif.html`, 1998. (p 5)

[KL89] M. Kifer and G. Lausen. F-Logic: A higher-order language for reasoning about objects, inheritance and scheme. In *Intl. Conf. on Very Large Data Bases (VLDB)*, pp. 134–146, 1989. (p 64)

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, July 1995. (pp 2, 63, 64, 67, 115, 136, 137, 187, 193, 206, 207)

[KS95] D. Konopnicki and O. Shmueli. W3QS: A Query System for the World-Wide Web. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 54–65, Zürich, Switzerland, 1995. (p 206)

[KS98] D. Konopnicki and O. Shmueli. Information Gathering in the WWW: The W3QL Query Language and the W3QS system. *ACM Transactions on Database Systems (TODS)*, 23(4):369–410, 1998. (p 206)

[KWD97] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper Induction for Information Extraction. In *Intl. Joint Conference on Artificial Intelligence*, 1997. (p 207)

[LHL+98] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective. *Information Systems*, 23(8):589–612, 1998. (pp 3, 64, 115, 131, 187, 190, 193, 207)

[LM01] G. Lausen and P. J. Marròn. HLCaches: An LDAP-based Distributed Cache Technology for XML. Technical Report 147, Universität Freiburg, Institut für Informatik, 2001. (p 184)

[LoP] The LoPiX System. `http://www.informatik.uni-freiburg.de/~may/lopix/`. (pp 72, 120, 193, 199, 214)

[LSS96a] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. A Declarative Language for Querying and Restructuring the Web. In *Proc. 6th Intl. Workshop on Research Issues in Data Engineering (RIDE)*, 1996. (p 206)

[LSS96b] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL – A Language for Interoperability in Relational Multi-Database Systems. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 239–250, 1996. (pp 66, 115)

[MAG+97] J. McHugh, S. Abiteboul, R. GOldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, 1997. (pp 208, 210)

[May99a] W. May. Information Extraction and Integration with FLORID: The MONDIAL Case Study. Technical Report 131, Universität Freiburg, Institut für Informatik, 1999. Available from `http://www.informatik.uni-freiburg.de/~may/Mondial/`. (pp 3, 163, 187, 207, 237)

[May99b] W. May. Modeling and Querying Structure and Contents of the Web. In *Workshop on Internet Data Modeling (IDM'99), Proc. DEXA 99 Workshops*, pp. 721–725. IEEE Computer Science Press, 1999. (p 190)

[May00a] W. May. Handling XML with FLORID, 2000. Available from `http://www.informatik.uni-freiburg.de/~dbis/florid`. (p 195)

[May00b] W. May. LoPiX: Logic Programming in XML: User Manual, 2000. Available from `http://www.informatik.uni-freiburg.de/~dbis/lopix`. (p 199)

[May01a] W. May. Information Integration in XML: The MONDIAL Case Study. Technical report, 2001. Available from `http://www.informatik.uni-freiburg.de/~may/lopix/lopix-mondial.html`. (pp 3, 163, 237)

[May01b] W. May. Integration of XML Data in XPathLog. In *CAiSE Workshop "Data Integration over the Web" (DIWeb'01)*, 2001. (p 2)

[May01c] W. May. XPathLog: A Declarative, Native XML Data Manipulation Language. In *International Database Engineering and Applications Workshop (IDEAS'01)*. IEEE Computer Science Press, 2001. (p 2)

[MHLL99] W. May, R. Himmeröder, G. Lausen, and B. Ludäscher. A Unified Framework for Wrapping, Mediating and Restructuring Information from the Web. In *International Workshop on the World-Wide Web and Conceptual Modeling (WWWCM)*, LNCS, no. 1727, pp. 307–320, 1999. (pp 131, 187, 207)

[MK98] W. May and P.-T. Kandzia. Nonmonotonic Inheritance in Object-Oriented Deductive Database Languages. Technical Report 114, Universität Freiburg, Institut für Informatik, 1998. Available from `http://www.informatik.uni-freiburg.de/~dbis/Publications/98/Inheritance%.html`. (pp 135, 136)

[MK01] W. May and P.-T. Kandzia. Nonmonotonic Inheritance in Object-Oriented Deductive Database Languages. *Journal of Logic and Computation*, 11(4), July 2001. (pp 135, 136, 137)

[MMM97] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the World Wide Web. *Intl. Journal on Digital Libraries (JODL)*, 1(1), 1997. (p 205)

[Moe00] G. Moerkotte. YAXQL: A powerful and web-aware query language supporting query reuse. Available at `http://pi3.informatik.uni-mannheim.de/staff/mitarbeiter/moerkotte/myself.html`, January 2000. (pp 19, 61, 63, 125)

[Mon] The MONDIAL Database. `http://www.informatik.uni-freiburg.de/~may/Mondial/`. (pp 3, 9, 10, 69)

[NCS93] NCSA Mosaic. Mosaic: An Internet Information Browser, 1993. `http://www.ncsa.uiuc.edu/SDG/Software/Mosaic`. (p 6)

[Ope] OpenLDAP. Lightweight Directory Access Protocol. `http://www.openldap.org/`. (p 7)

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. MedMaker: A Mediation System Based on Declarative Specifications. In *Intl. Conference on Data Engineering (ICDE)*, pp. 132–141, 1996. (p 208)

[Poo94] D. Poole. Default Logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming* [GHR94]. (p 136)

[Prz88] T. C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, ed., *Foundations of Deductive Databases and Logic Programming*, pp. 191–216. Morgan Kaufmann, 1988. (p 119)

[PV99] Y. Papakonstantinou and P. Velikhov. Enhancing Semistructured Data Mediators with Document Type Definitions. In *Intl. Conference on Data Engineering (ICDE)*, 1999. (p 210)

[RAH+96] M. T. Roth, M. Arya, L. Haas, M. Carey, W. Cody, R. Fagin, P. Schwarz, J. Thomas, and E. Wimmers. The Garlic Project. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1996. (p 209)

[RCF00] J. Robie, D. Chamberlin, and D. Florescu. Quilt: an XML Query Language. `http://www.almaden.ibm.com/cs/people/chamberlin/quilt.html`, 2000. (pp 18, 54, 55)

[RDF00] Resource Description Framework (RDF). `http://www.w3.org/RDF`, 2000. (p 183)

[RLS98] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). QL'98 - The Query Languages Workshop; `http://www.w3.org/TandS/QL/QL98/pp/xql.html`, 1998. (pp 17, 28, 29, 70, 121)

[RML01] Rule Markup Language. `www.dfki.uni-kl.de/ruleml/`, 2001. (p 130)

[Rob99] J. Robie. XQL (XML Query Language). `http://www.metalab.unc.edu/xql/xql-proposal.html`, 1999. (pp 1, 17, 28, 29, 121)

[SA99] A. Sahuguet and F. Azavant. Looking at the Web through XML glasses. In *Intl. Conference on Cooperative Information Systems (CoopIS)*. IEEE Computer Science Press, 1999. (p 206)

[Sah00] A. Sahuguet. Kweelt, the Making-of: Mistakes made and Lessons Learned. Technical report, Univ. of Pennsylvania, 2000. available at `db.cis.upenn.edu/Kweelt`. (pp 19, 27, 58, 209)

[SGT+99] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. J. D. Witt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Intl. Conference on Very Large Data Bases (VLDB)*, 1999. (pp 110, 211)

[SM] C. M. Sperberg-McQueen. Bison grammars and Flex scanners for SGML document type definitions. `ftp://ftp-tei.uic.edu/pub/tei/sgml/grammar/bison/`. ACH/ACL/ALLC Text Encoding Initiative University of Illinois at Chicago. (p 195)

[Sof] Software AG. Tamino – An Internet Database System. `http://www.tamino.com/`. (pp 7, 17, 19, 30, 62, 210)

[Suc97] D. Suciu, ed. *Proc. of the Workshop on Management of Semi-Structured Data (in conjunction with SIGMOD/PODS)*, Tucson, Arizona, 1997. `http://www.research.att.com/~suciu/workshop-papers.html`. (p 5)

[Tar99] P. Tarau. Jinni: Intelligent Mobile Agent Programming at the Intersection of Java. In *PAAM*, 1999. `www.binnetcorp.com/Jinni`. (p 207)

[Tau] J. Tauber. FOP: An Open-Source XSL Formatter and Renderer. `http://www.jtauber.com/fop/`. (p 30)

[TIHW01] I. Tatarinov, Z. G. Ives, A. Halevy, and D. Weld. Updating XML. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 2001. (pp 1, 13, 19, 59, 60, 63, 113, 125, 126, 129, 211, 212, 214)

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Heterogeneous Databases and the Design of Disco. In *Int. Conf. on Distributed Computing Systems*, pp. 449–457, 1996. (p 209)

[W3C] W3C – The World Wide Web Consortium. `http://www.w3c.org/`. (p 17)

[Wad99a] P. Wadler. A formal semantics of patterns in XSLT. In *Markup Technologies*, 1999. `http://www.cs.bell-labs.com/who/wadler/topics/xml.html`. (pp 17, 25)

[Wad99b] P. Wadler. Two semantics for XPath. 1999. `http://www.cs.bell-labs.com/who/wadler/topics/xml.html`. (pp 17, 25, 26, 71, 80, 82, 83, 84, 85, 86, 87, 88, 97, 246)

[Weh01] D. Wehr. Evaluierung des Excelon XML Datenbanksystems. Studienarbeit, Universität Freiburg (in german), 2001. (p 201)

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 1992. (pp 205, 207, 210)

[XLi00] XML Linking Language (XLink). `http://www.w3.org/TR/xlink`, 2000. (pp 45, 50, 177, 180)

[XML98] Extensible Markup Language (XML). `http://www.w3.org/XML/`, 1998. (pp 6, 7)

[XML99a] XML Schema. `http://www.w3.org/XML/Schema`, 1999. (pp 14, 18, 39, 145)

[XML99b] XML Schema Part 1: Structures. `http://www.w3.org/TR/xmlschema-1`, 1999. (p 148)

[XML99c] XML Schema Part 2: Datatypes. `http://www.w3.org/TR/xmlschema-2`, 1999. (pp 39, 160)

[XMQ01a] XML Query Algebra. `http://www.w3.org/TR/query-algebra`, 2001. (pp 18, 50, 52, 54, 67, 126)

[XMQ01b] XML Query Data Model. `http://www.w3.org/TR/query-datamodel`, 2001. (pp 18, 37, 50, 51, 59, 60, 75)

[XMQ01c] XML Query Requirements. `http://www.w3.org/TR/xmlquery-req`, 2001. (pp 18, 19, 38, 63, 129)

[XPa99] XML Path Language (XPath). `http://www.w3.org/TR/xpath`, 1999. (pp 1, 17, 20, 22, 69, 70, 81, 82, 83)

[XPt00] XML Pointer Language (XPointer). `http://www.w3.org/TR/xptr`, 2000. (pp 17, 45, 50, 177)

[XQu01] XQuery: A Query Language for XML. `http://www.w3.org/TR/xquery`, 2001. (pp 1, 17, 19, 27, 54, 58, 59, 72, 178)

[XSL98] Extensible Stylesheet Language (XSL). `http://www.w3.org/Style/XSL/`, 1998. (p 17)

[XSL99] XSL Transformations (XSLT). `http://www.w3.org/TR/xslt`, 1999. (pp 1, 17, 30, 31, 178)

Curriculum Vitae

Personal Data

Date and place of birth:	24.5.1968 in Giessen/Lahn (Germany)
Nationality:	german

Professional

6.12.2001	Habilitation (venia legendi) in Computer Science. Thesis: *"A Logic-Based Approach to XML Data Integration"*.
Since 1.1.1999	Scientific Assistant (*Wissenschaftlicher Assistent, C1*) in the database research group at Universität Freiburg (Germany).
17.11.1998	PhD (Dr. rer. nat.) in Computer Science from Universität Freiburg; grade "sehr gut".
1.11.97–31.12.98	Research Assistant (*Wissenschaftlicher Mitarbeiter*) in the database research group at Universität Freiburg.
1.5.95–31.10.97	Scholarship in the graduate program *Menschliche und Maschinelle Intelligenz (Human and Artificial Intelligence)* at Universität Freiburg, associated to the database research group.

Studies

10.89–2.95	Studies in Computer Science at Universität Karlsruhe (TH), Degree (Dipl.-Inform.) passed with distinction. The diploma thesis *Protocol Verification in Temporal Logic* was awarded the *Förderpreis des FZI (Research Center in Computer Science).*
10.91–3.93	Teaching Assistant for basic computer science courses *Informatik I-III.*

Civil Service

6.87–1.89	Civil service on a surgical ward at the town hospital in Furtwangen.

School

15.5.1987	Abitur, overall grade 1,2 (best of that year).
1978–1987	Secondary school (Gymnasium) in Furtwangen.
1974–1978	elementary school in Furtwangen/Black Forest.

Languages

German:	native
English:	fluent
French:	good
Swedish:	fairly good
Russian:	basic knowledge