Querying Semistructured Data Using a Rule-Oriented XML Query Language

Tadeusz Pankowski[1]

Abstract. The goal of the paper is to propose a semistructured data model for representing XML documents and a language for querying semistructured database representing XML data. The language is based on a path calculus and on its extension involving rules (in Datalog style) and Skolem functions. Three kinds of matching between query variables and database objects are discussed: a rigid, semirigid, and flexible matching. The flexible matching is of special importance since semistructured data does not conform to a rigid schema, its structure is often not known in advance and its structure may change frequently. We propose a method, which is based on regular path expressions, supporting valuation of query variables according to those three kinds of matching. The containment problem for this matchings is discussed. The main idea of an experimental implementation is outlined.

1 INTRODUCTION

Semistructured data has no absolute schema fixed in advance, has irregular structure, is incomplete and heterogeneous. Such data arises mainly when the data is stored in sources that do not require a rigid structure (such as the World-Wide Web), and when data is integrated from heterogeneous sources (especially when new sources are frequently added). It is commonly agreed that data models and query languages, designed for well-structured data, are inappropriate in such environments [1], [2]. XML (Extensible Markup Language) [3] is fast becoming the dominant standard for representation and interchange semistructured data on the Internet. Many heterogeneous information sources can structure their external views as repositories of XML data, no matter what their internal storage mechanisms.

Thus, modeling and querying semistructured data repositories is a new challenge in the field of intelligent data retrieval and knowledge-based systems.

In the paper we follow the assumption that semistructured data is a rooted, labeled directed graph. Such data must be:

- exchanged over the Web and presented to users,
- stored in some repositories - in text files on the Web, within a database system supporting storing and querying the data, or in a mixed form,
- capable for reasoning, especially when heterogeneous data sources are integrated or when approximate answers to queries are expected.

We discuss some modeling alternatives to achieve goals mentioned above. For exchanging and presenting data - the XML data model, for storing the data within conventional database system - an object-oriented model being a variant of OEM [1], [4], and for reasoning purposes - a very simple description logic (to show that a semistructured data can be understood as a knowledge base) [5], [10].

The main part of the paper concerns querying semistructured data. Since the same information may be captured by diversity of data structures, a fixed query and exact matching (that is typical for well-structured data) could not give the all expected information. To solve this problem, and to avoid query rewriting, in [6] a flexible matching is proposed. We follow this idea and propose a class of path expressions as well as an algorithmic method for computing all ”standard” and ”non-standard” matchings for such path expressions. Next we propose a path calculus. Path calculus queries return answers consisting of tuples of objects. To construct well-formed semistructured data from answers, we extend the query language introducing rules. Path calculus query forms the body of a rule and the rule head is devoted to create the expected resulting object. Rules are in Datalog style and involve Skolem functions.

The main contribution of the paper is the original proposal of well-defined syntax and semantics of a language for manipulating semistructured data. The rule-orientation of the language allows for utilization of both the paradigm of first-order calculus and the flexibility of data structuring in XML. Regular path expressions support three kinds of matching, the rigid matching (typical for well-structured data query languages), semirigid matching (typical for other semistructured data languages) and flexible matching. The problem of query containment for queries based on those three kinds of matching is discussed.

The rest of the paper is organized as follows: Section 2 gives basic ideas and motivation for research. Models of semistructured data are discussed in Section 3. In Section 4 we give syntax and semantics of path calculus and its rule-oriented extension. A theorem of query containment is proven. Section 5 concludes the paper and outlines some future research.

2 MOTIVATION AND BASIC IDEA

Semistructured data retrieval systems may be characterized as follows [6]:

- data does not conform to a rigid schema - it is difficult to design queries,
- the structure of the database changes frequently - queries should be rewritten frequently,
- data is contributed by many users in a variety of design - the query should deal with different structures of data,
- the description of the schema is large (e.g. made in DTD) - it is difficult to use the schema for formulating queries.

[1] Chair of Control, Robotics and Computer Science, Poznań University of Technology, Poland, email: Tadeusz.Pankowski@put.poznan.pl

Semistructured data is commonly perceived as a rooted, labeled directed graph. All edges, all leaves and some non-leaf nodes of a data graph are labeled with strings (texts). Additionally, a unique identifier may be assigned to each node (we will denote identifiers by integers). There is a *root label*, which uniquely identifies a data graph. An example of a data graph is in Figure 1, *bib* is the root label of the data graph.

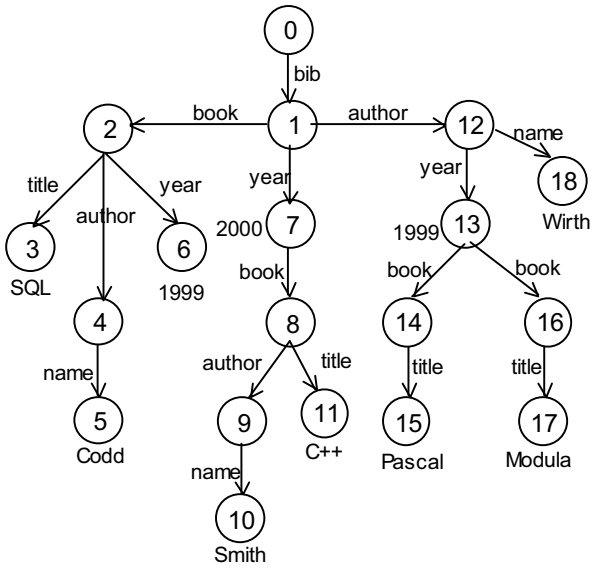

Figure 1. An example of data graph

Semistructured query languages have ability to reach to arbitrary depth in the data graph. This is possible by means of path expression. A simple path expression is a sequence of edge labels separated by dots and starting with the root label. E.g. the following path expression

$$bib.book.author.name$$

is intended to look for all the books and their authors. Matching is a mapping which assigns a sequence of data graph edges to every path expression. A label L from a path expression is mapped to a data graph edge with label L. Every matching of a path expression $L_1, ..., L_n$ can be represented as a sequence of nodes $(i_1, ..., i_n)$, where i_k is a node with an incoming edge labeled with L_k. In exact or rigid matching adjacent path expression labels are mapped to adjacent data graph edges. Notice that using the exact matching, we are not able to find all answers because an author node can either be below or above a book node.

The classical query-answer paradigm can not be directly used for querying semistructured data. This is due to the following:

1. To capture irregular structure of data and to avoid query reformulation, an extended concept of matching path expressions and data graph paths is needed. We distinguish three kinds of matching:
   - in the *rigid matching* adjacent labels of a path expression are mapped to adjacent edges of one path and the ordering is preserved, e.g. (1, 2, 4, 5) rigidly matches the path expression *bib.book.author.name*, while (1, 8, 9, 10) does not;
   - in the *semirigid matching*, labels of a path expression are mapped to edges of one path and the ordering is preserved, but adjacent labels in the path expression must not be mapped to adjacent edges of the database path, e.g. (1, 8, 9, 10) semirigidly matches *bib.book.author.name*, while (1, 14, 12, 18) does not;
   - in the *flexible matching* labels of a path expression are mapped to edges of data graph; the ordering need not be preserved; notice that the path expression is not necessarily mapped to a path in the data graph. E.g. (1, 14, 12, 18) flexibly matches the path expression *bib.book.author.name*. Because such notion of acceptable matchings (answers) might be too general, we will assume some restrictions. In some approaches the problem of choosing appropriate answers is solved by means of ranking them (e.g. [7]).
2. For lack of basic predicates (like relation names in relational calculus) specified in advance, some partially defined path expressions may be used. In our approach, any path expression is consistent with some matching type.
3. Final result of the query must be provided as an object (e.g. as an XML document). Thus, a mechanism for creating new objects should exist in the query language.

3 SEMISTRUCTURED DATA MODELS

Semistructured data is commonly specified as an XML document. Thus we assume that a system for processing semistructured data should support XML format both on its input and output. In our experimental implementation [13] XML data is transformed into an object-oriented model and is stored within a relational database system. In this section we review the modeling capability of XML, propose a semistructured object-oriented data model (SSOO), and show how semistructured data can be defined by means of a description logic.

3.1 XML as a model of semistructured data

An XML document is a textual representation of data and consists of hierarchically nested element structure starting with a root element. The basic component in XML is an *element*. An element consists of *start-tag*, the corresponding *end-tag* and *content*, i.e. the structure between the tags. All content elements are *subelements* of their parent element. With elements we can associate *attributes*. An attribute is a pair $(name = "value")$ and is included within the start-tag of an element. A well-formed XML element must conform to the following syntax (* denotes zero or many occurrences of the preceding symbol):

element	::=	<tag *attribute**> *content**</tag>
attribute	::=	name="text"
content	::=	*element* | text

The semistructured data from Figure 1 can be specified as the following XML data:

```
<bib>
  <book title="SQL" year="1999">
    <author>
      <name>Codd</name>
    </author>
  </book>
  <year>
    2000
    <book title="C++">
      <author>
```

```
        <name>Smith</name>
      </author>
    </book>
  </year>
  <author name="Wirth">
    <year>
      1999
      <book>
        <title>Pascal</title>
      </book>
      <book>
        <title>Modula</title>
      </book>
    </year>
  </author>
</bib>
```

3.2 Object-oriented model of semistructured data

Object-oriented approaches to semistructured data models were emerging with the development of the Object Exchange Model (OEM) [4], [8] and are rooted in object-oriented database models. There is a number of variants of the OEM data model [1]. In this paper we propose a variant called SSOO (+ denotes one or many occurrences of the preceding symbol):

object ::= oid (*content*+)
content ::= label : oid | text

Definition 1 *Let ID be a countable and infinite set of object identifiers, LAB - a set of labels, TXT - a set of texts, and CONT - a set of contents created according to the rules specified above.*

A quadruple DB = (ID, LAB, TXT, CONT) we call a database, and a pair (OID, con) - a state of the databas, where

- *OID is a finite subset of ID,*
- $con : OID \rightarrow 2^{CONT}$ *is a function which associates a subset of CONT to every identifier in OID.*

An identifier $id \in ID$ is said to be defined if con is defined on it; if id is not defined we assume $con(id) = \emptyset$. Additionally, we assume that every identifier occurring in con(id), for $id \in OID$, is defined. □

3.3 Modeling semistructured data in description logic

Description logics allow for representing a domain of discourse in terms of *concepts* and *roles*. Concepts model classes of individuals, while roles model relationships between classes. Starting from *atomic concepts* and *atomic roles*, one can build complex concepts and roles by applying certain *constructs*. We will consider a very simple description logic, SSDL, which can be used to describe semistructured data. In SSDL three constructors are assumed: *existential quantifiers over roles* (∃), *intersection on concepts* (⊓), and *role inverse* (!). We use the following notations: L for *atomic roles*, A for *primitive atomic concepts*, and D for *defined atomic concepts*. Complex concepts, C, and complex roles, R, conform to the following syntax:

C ::= $\exists R.D \mid A \mid C \sqcap C$,
R ::= $L \mid !L$.

Semantics of SSDL might be defined by means of interpretation $I = (\Delta^I, \cdot^I)$, where Δ^I is a set of individuals, and $\cdot^I$ is an *interpretation function*. Then:

$$\begin{aligned} A^I &\subseteq \Delta^I, \\ D^I &\subseteq \Delta^I, \\ L^I &\subseteq \Delta^I \times \Delta^I, \\ !L^I &= \{(x', x) | (x, x') \in L^I\}, \\ (\exists R.D)^I &= \{x \in \Delta^I | \exists x'.(x, x') \in R^I \wedge x' \in D^I\}, \\ (C \sqcap C')^I &= C^I \cap C'^I. \end{aligned}$$

A set of assertions (definition statements) of the form

$$D =_{def} C$$

is a *semistructured knowledge base*. In a definition statement (see [9]) D cannot appear in the left-hand side of other definition statements. The definition is intended to define the concept D in terms of C, i.e., to define D as a concept denoting the set of the individuals satisfying C. E.g. the concept: $1999 \sqcap \exists author.C5$ may denote individuals that are from the year 1999 and that are related through the binary relation (role) $author$ with some individual from the set of individuals denoted by the concept $C5$.

To describe SSOO data by means of SSDL, we assume that texts are primitive concepts, identifiers are defined concepts, and labels are roles. Then for data from Figure 1, we have: *C0 = ∃ bib.C1, C1 = ∃ book.C2 ⊓ ∃ year.C7 ⊓ ∃ author.C12, ..., C7 = 2000 ⊓ ∃ book.C8,* ..., etc.

4 QUERY LANGUAGE

Our query language ([13], [15]) is based on regular paths which are the basic constructs for semistructured data query languages [8], [11] [12]. The differences are what a path expression denotes. In our approach, it denotes a set of oid sequences (paths) of the same length. The set is called a *path table* and is an extension of the path expression. Components of the path expression, separated by dots, constitute the *type* of the path table. This approach was used in an experimental system, called XML-SQL, which we implemented in a relational database system [13].

In this section we mainly address the problem of facilitating different kinds of matchings. We propose a class of path expressions and rules for computing their extensions (semantics). The extension of a path expression consists of a set of matchings, and the structure of the path expression determines the kind of matching.

4.1 Path calculus

Path expressions:

P ::= $L \mid P.E \mid P.(*, !\_).E \mid P.(!\_, *).E$
E ::= $L \mid * \mid !\_$

A path expression P consists of identity expressions, E, where every identity expression returns a set of object identifiers. The value of a path expression which consists of n-components ($n \geq 1$) separated by dots, is a set of sequences of object identifiers of length equal to n. The meaning of identity expression is:

- label L denotes all $oids$ of objects (nodes) with an incoming edge labeled with L, every path expression must begin with the root label,
- "$*$" matches any path (also the empty path),
- "!_" matches inverse of any edge,
- $P.(*, !\_).E$ and $P.(!\_, *).E$ are conditional path expressions.

Semantics of path expressions:

To define semantics for path expressions we assume that a data graph is represented by means of the following two tables:

EdgeTab(ParentOid, Label, Oid).
ValueTab(ParentOid, Value).

If $oid1(content)$ is an object, where $(L : oid2) \in content$, and $text \in content$, then:

$(oid1, L, oid2) \in EdgeTab$,
$(oid1, text) \in ValueTab$.

Every path expression P is treated as an 1-place predicate denoting a set of *oid* sequences. Semantics of path expressions is given by the following rules; $last(v)$, and $front(v)$ returns, respectively, the last element, and the front (all elements except the last one) of a sequence v; $u \bullet w$ denotes the concatenation of sequences u and w:

1. $"L"(x)$:- $EdgeTab(\_, L, x)$
2. $"P.L"(v)$:- $"P"(u) \wedge EdgeTab(last(u), L, x) \wedge v = u \bullet x$
3. $"P.*"(v)$:- $"P"(u) \wedge v = u \bullet last(u)$
   $"P.*"(v)$:- $"P.*"(u) \wedge EdgeTab(last(u), \_, x) \wedge v = front(u) \bullet x$
4. $"P.!\_"(v)$:- $"P"(u) \wedge EdgeTab(x, \_, last(u)) \wedge v = u \bullet x$
5. $"P.(*,!\_).E"(v)$:- $"P.*.E"(v)$
   $"P.(*,!\_).E"(v)$:- $\neg("P.*.E"(v) \wedge "P.(!\_,*).E"(v)$
6. $"P.(!\_,*).E"(v)$:- $"P.!\_.E"(v)$
   $"P.(!\_,*).E"(v)$:- $\neg"P.!\_.E"(v) \wedge "P.!\_.*.E"(v)$
   $"P.(!\_,*).E"(v)$:- $\neg"P.!\_.E"(v) \wedge \neg("P.!\_.*.E"(v) \wedge "P.!\_.(!\_,*).E"(v)$

Definition 2 *A query over a DB is an expression of the form*

$$\{path\_var.E, ..., path\_var.E | \phi\},$$

where ϕ is a formula (query qualifier), $path\_var$ is a path variable, and E is an identity expression. The syntax of a query qualifier is:

$$\begin{array}{rcl} \phi & ::= & "P"(path\_var) \mid \neg\phi \mid \phi \wedge \phi \mid \exists path\_var\ \phi \\ & & \mid scalar = scalar, \\ scalar & ::= & path\_var.E \mid text \mid val(path\_var.E) \end{array} \quad \square$$

The following expression is a query:

$$\{v.title \mid "bib.*.book.title"(v) \wedge \exists u "bib.*.book.(*,!\_).year"(u) \wedge v.book = u.book \wedge val(u.year) = 1999\}.$$

Example 1 *To find all books and their authors, we can write the following queries:*

Q_1 = {*v.book, v.author, v.name* | *"bib.book.author.name"(v)*},
Q_2 = {*v.book, v.author, v.name* | *"bib.*.book.author.name"(v)*},
Q_3 = {*v.book, v.author, v.name* | *"bib.*.book.(*, !_).author.name"(v)*},

where results for these queries against Figure 1 are, respectively:

Q_1 = {*(2, 4, 5)*},
Q_2 = {*(2, 4, 5), (8, 9, 10)*},
Q_3 = {*(2, 4, 5), (8, 9, 10), (14, 12, 18) (16, 12, 18)*}. $\square$

Each of the above queries is intended to retrieve the same information but uses different semantics for matching. This semantics is chosen through the path expression used in qualifier of the query (rigid, semirigid and flexible). The consequences of making the decision can be formalized as follows.

Definition 3 *Let P be a path expression and $l_1, ..., l_n$ be all labels occurring in P. A ground query over a database DB and a path expression P is a query of the form*

$$Q_{DB}(P) = \{v.l_1, ..., v.l_n | P(v)\}. \quad \square$$

Proposition 1 *Let P be a path expression composed of two path expressions P_1 and P_2, i.e. $P = P_1.P_2$. Then for every database DB the following containments hold*

$$Q_{DB}(P_1.P_2) \subseteq Q_{DB}(P_1.*.P_2) \subseteq Q_{DB}(P_1.(*,!\_).P_2),$$

where each $Q_{DB}(P)$ is a ground query over DB and P.

Proof To show the first inclusion let us assume that
$"P_1.P_2"(v)$, $v = v_1 \bullet v_2$ and $P_1(v_1) \wedge P_2(v_2)$.
Then for $v' = v_1 \bullet last(v_1) \bullet v_2$, the formula

$$"P_1.*.P_2"(v')$$

holds. So, the first inclusion is true.

The second inclusion follows from the implication:

$$"P_1.*.P_2"(v) \Rightarrow "P_1.(*,!\_).P_2"(v).$$

Thus, the proposition is proven. $\square$

A query $Q_{DB}(P)$ is a rigid query if it has no occurrence of $*$ and $!\_$; it is a semirigid query if it has not any occurrence of $!\_$. Otherwise it is a flexible query. For our running example we have:

$$Q_1 \subseteq Q_2 \subseteq Q_3$$

Q_1, Q_2, and Q_3 are rigid, semirigid and flexible ground queries, respectively. Notice that Q_3 is equivalent to the query:

$$Q_4 = \{v.book, v.author, v.name | "bib.*.author.name.(*,!\_).book"(v)\}$$

Two databases DB and DB' are equivalent if for any query Q,

$$Q_{DB} = Q_{DB'}.$$

The equivalence may be referred to as a rigid, semirigid, or flexible equvalence, depending on the class of queries under consideration.

4.2 Creating objects

To support creation of objects we use rules, where head of a rule constructs a semistructured data object, and the body of the rule delivers bound variables used in the construction. A rule conforms to the following syntax:

$$obj\_expr \quad :- \quad Query(x_1, ..., x_n)$$

where

$$\begin{array}{rcl} obj\_expr & ::= & oid\_expr \mid oid\_expr(content\_expr+) \\ content\_expr & ::= & label\_expr : obj\_expr \mid text\_expr \\ oid\_expr & ::= & newOid(variable*) \mid variable \end{array}$$

A rule is intended to create a new object (answer to a query). The head of the rule constructs the object using bound variables defined in the body. The body of the rule is a formula $Query(x_1, ..., x_n)$, where Q is a query considered as an n-place predicate, $n \geq 1$. Such defined variables are then used as control variables or parameters within the head of the rule.

The $newOid(x_1, ..., x_n)$ function is a *Skolem function* (in the context of the head it occurs in). Every invocation of $newOid()$ without arguments creates a new object and returns its identifier. The function with arguments $x_1, ..., x_n$, creates a new object for every distinct valuation of variables, we say that this object and its identifier depend on $x_1, ..., x_n$. Every next invocation for the same valuation of arguments returns the identifier that depends on this valuation.

The semantics of the expression $newOid(...)(content\_expr)$ is as follows:

- if $newOid()$ has the empty set of arguments or it is the first invocation of $newOid(...)$ with the given arguments, then a new object is created with the content determined by the $content\_expr$ and the identifier of the newly created object is returned;
- every next invocation of $newOid(...)(content\_expr)$ with the same values of arguments appends the content determined by the $content\_expr$ to the current content of the object created by the first invocation; the identifier of the object is returned.

Formally, every invocation of

$$newOid(...)(content)$$

in a database state $s = (OID, con)$ returns an identifier id and changes state s into $s' = (OID', con')$ where:

$$\begin{array}{rcl} OID' & = & OID \cup \{id\}, \\ con'(i) & = & con(i), \text{ for } i \neq id, \\ & & \text{and } con'(id) = con(id) \cup content. \end{array}$$

Example 2 *Let Q be a query from Example 1. By means of the rule, the following result object may be constructed:*

```
newOid()(result :
  newOid($b)(book:$b,
    newOid($b,$a)(author:$a)
  )
) :- {v.book,v.author,v.name|
"bib.*.book.(*,!_).author.name"(v)} ($b,$a,$n)
```

This query can be written as the following XML-SQL rule [13]:

```
<result>
  <book>($b)
    $b
    <author>($b,$a)
      $n
    </author>
  </book>
</result> :- Answer($b,$a,$n)
```

where the Answer *table is obtained by the following SQL query operating on path tables:*

```
select v.book as $b,
       v.author as $a,
       v.name as $n into Answer
from "bib.*.book.(*,!_).author.name" v
```

The resulting object from Example 2 can be in turn introduced into the database, so the database is transformed into a new state. Note, however, that some book objects (e.g. 2 and 8) may contain information about authors. In such a case we say that the book object subsumes their author object. A subsumed object should be removed from the resulting object to avoid unnecessary redundancy. (Subsumptions between semistructured objects we have discussed in [14]).

5 CONCLUSION

In the paper we have discussed some problems concerning querying semistructured data repositories. The role of such information sources in data retrieval and knowledge-based systems becomes increasingly important. We consider a semistructured data model in which a semistructured database is perceived as an edge labeled graph. Data retrieval is performed by means of a first-order path calculus. To capture non-rigid data, we propose a method that allows to use rigid, semirigid or flexible semantics. The query containment problem for these semantics is discussed. Modifications of the database, constructing query results as objects and inserting them into the database is supported by the rules and Skolem functions [15]. There are several interesting problems to investigate within our approach. One is to study implications due to the fact that operating on path tables we actually operate on views over the database. The other problem concerns object subsumptions and their consequences to data retrieval and result construction.

REFERENCES

[1] S. Abiteboul, P. Buneman, D. Suciu, *Data on the Web. From Relational to Semistructured Data and XML*, Morgan Kaufmann, San Francisco, 2000.

[2] V. Vianu, A Web Odyssey: from Codd to XML, *Proc. of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS'2001, May 21-23, Santa Barbara, 2001.

[3] *Extensible Markup Language (XML)*, World Wide Web Consortium (W3C), www.w3.org.TR/REC-xml.

[4] Quass D., Rajaraman A., Sagiv Y., Ullman J., Widom J., Querying Semistructured Heterogenous Information, *Proc. of International Conference on Deductive and Object-Oriented Databases, DOOD95*, LNiCS 1013, 319–344, 1995.

[5] D. Calvanese, G. De Giacomo, M. Lenzerini, Modeling and Querying Semi-Structured Data, *Networking and Information Systems Journal* **2** (2), 253–273, 1999.

[6] Y. Kanza, Y. Sagiv, Flexible Queries over Semistructured Data, *Proc. of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS'2001, May 21-23, Santa Barbara, 2001.

[7] S.Amer-Yahia, S. Cho, D. Srivastava, Tree Pattern Relaxation, *Advances in Database Technology - EDBT 2002*, (C. S. Jensen, K. G. Jeffery, J. Pokorny, S. Saltenis, E. Bertino, K. Bloehm, M. Jarke, eds.), LNiCS 2287, Springer, 496–513, 2002.

[8] Abiteboul S., D. Quass, J. McHugh, J. Widom, J.L. Wiener, The Lorel query language for semistructured data, *International Journal of Digital Libraries* **1** (1), 68–88 (1997).

[9] G. De Giacomo, Lenzerini M., A Uniform Framework for Concept Definitions in Description Logics, *Journal of Artificial Intelligence Research* **6**, 87–110 (1997).

[10] I. Horrocks,DAML+OIL: A Reason-able Web Ontology Language, *Advances in Database Technology - EDBT 2002*, (C. S. Jensen, K. G. Jeffery, J. Pokorny, S. Saltenis, E. Bertino, K. Bloehm, M. Jarke, eds.), LNiCS 2287, Springer, 2–13, 2002.

[11] P. Buneman, M. Fernandez, D. Suciu, UnQL: a query language and algebra for semistructured data based on structural recursion, *The VLDB Journal* **9**, 76–110 (2000).

[12] XQuery 1.0: An XML Query Language. *W3C Working Draft*, www.w3.org/TR/xquery

[13] T. Pankowski, XML-SQL: An XML Query Language Based on SQL and Path Tables, *VIII. Conference on Extending Database Technology, EDBT'2002, Workshop XML-Based Data Management (XMLDM)*, March 24-28, Prague, 48–68, 2002.

[14] T. Pankowski, Approximate Answers in Databases of Labeled objects, *Intelligent Information Systems* , (M. Klopotek, M. Michalewicz, S.T. Wierzchon, eds), Advances in Soft Computing, Physica-Verlag A Springer-Verlag Company, Heidelberg New York, 351–361, 2000.

[15] T. Pankowski, PathLog: A query language for schemaless databases of partially labeled objects, *Fundamenta Informaticea*, **49** 4, 369–395, 2002.