XML Document Correction and XQuery Analysis with *Analyzer*

Jakub Stárka, Martin Svoboda, Jiří Schejbal, Irena Mlýnková, and David Bednárek

Department of Software Engineering, Faculty of Mathematics and Physics
Charles University in Prague
Malostranské náměstí 25, 118 00 Praha 1, Czech Republic
{starka, svoboda, mlynkova, bednarek}@ksi.mff.cuni.cz

Abstract. This paper describes extensions of our previously proposed SW prototype – Analyzer, a framework for performing statistical analyses of real-world XML data. Firstly, it describes the design and implementation of a system for the analysis of collection of XQuery programs. It is based on the frequency of the occurrence of various language constructs and their combinations defined by the user. In the core of the system, the XQuery program is converted to a suitable XML representation which allows for analytical queries formulated in the XPath language. Secondly, we introduce the model involving repairs of elements and attributes with respect to single-type tree grammars. Via the inspection of the state space of an automaton recognising regular expressions, we are always able to find all minimal repairs represented by recursively nested multigraphs, which can be translated to particular sequences of edit operations altering data trees. We have proposed four particular algorithms and provided the prototype implementation supplemented with experimental results.

1 Introduction

The eXtensible Markup Language (XML) [5] is currently a de-facto standard for data representation. Its popularity is given by the fact that it is well-defined, easy-to-use and, at the same time, enough powerful. The problem is that the XML standards were proposed in full possible generality so that future users can choose what suits them most. Nevertheless, the real-world XML data are usually not so “rich”, thus the effort spent on every possible feature is mostly useless.

Exploitation of results of statistical analyses of real-world data is a classical optimization strategy in various areas of data processing. It is based on the idea to focus primarily on efficient implementation of constructs that are used in real-world data most often. One of the most important advantages of statistical analyses of real-world data is refutation of incorrect assumptions on typical use cases, features of the data, their complexity etc. As an example we can consider exploitation of recursion. The support for recursion is often neglected and it is considered as a side/auxiliary construct. However, analyses [11] show that in selected types of XML data it is used quite often and, hence, is efficient, or at least any support is very important. On the other hand, the number of distinct recursive elements is typically low (for each category less than 5) and that the type of recursion commonly used is very simple.

However, working with real-world data is not simple, because they can often change, are not precise, or even involve a number of errors. In this case we can either discard the incorrect data, and, hence, loose a significant portion of them, or provide a kind of *corrector.*

In the next step we want to make the analyses themselves. Currently there exists a number of papers which focus on statistical analyses of real-world XML data [11, 3, 10], however an analysis of real-world XML operations, in particular queries, is still missing. The reason is mainly the complexity of crawling a representative set and the complexity of the analytical process.

In this paper we describe two parts of a general framework for statistical analyses of real-world XML data called *Analyzer.* Firstly, we focus on a correction framework involving structural repairs of elements with respect to a single-type tree grammar. Secondly, we describe the usage of the framework for XQuery analysis. Since there is no standardized real-world datasets, we use two artifical collections to demonstrate the approach.

Outline The paper is structured as follows: In Section 2 we describe the related work, in particular concerning corrections of data. In Section 3, we describe the architecture of *Analyzer* which indicates its general functionality. Section 4 is devoted to processing of incorrect data. In Section 5, we show the principles of used query analysis and we show results of a query analysis of some artificial data. Finally, in Section 6 we conclude.

Relation to Previous Work In this paper, we extend our previous work [17]. Motivated by a successful and interesting statistical analysis of real-world XML data [11], *Analyzer* was implemented as a SW project of Master students of the Department of Software Engineering of the Charles University in Prague. Its installation package as well as documentation and source files can be found at its official web site [14]. Its first release 1.0 involved only basic functionality to demonstrate its key features and advantages and it was briefly introduced in paper [17]. In the following text, we describe extensions of *Analyzer* focused on XML document correction which was firstly proposed in [15] and extended in [16], and new module for XQuery analysis.

2 Related Work

As we have mentioned in the introduction, while currently there exists a number of papers focussing on statistical analysis of XML documents, XML schemas or their mutual comparison [11, 3, 10], there is no paper that would describe either the results or the process of analysis of real-world XML queries. Thus in this section we focus on the the related work of the second aim of this paper – correction of XML documents, in particular their re-validation.

The proposed correction model is based primarily on ideas from [2] and [13]. Authors of the former paper dynamically inspect the state space of a finite automaton for recognising regular expressions in order to find valid sequences of child nodes with minimal distance. However, this traversal is not effective, requires a threshold pruning to cope with potentially infinite trees, repeatedly computes the same repairs and acts efficiently only in the context of incremental validation. Although these disadvantages are partially handled in the latter paper, its authors focused on documents querying, but not repairing.

Next, we can mention an approximate validation and correction approach [18] based on testers and correctors from the theory of program verification. Repairs of data inconsistencies like functional dependencies, keys and multivalued dependencies are the subject of [12, 20].

Contrary to all existing approaches, we consider single type tree grammars instead only local tree grammars. Thus, we work both with DTD and XML Schema. Approaches in [2, 18] are not able to find repairs of more damaged documents, we are able to always find all minimal repairs and even without any threshold pruning to handle potentially infinite XML trees. Next, we have proposed much more efficient algorithm following only perspective ways of the correction and without any repeated repair computations. Finally, we have a prototype implementation available at [7] and performed experiments show a linear time complexity depending on a number of nodes in documents.

3 Framework Description

This section briefly concerns with *Analyzer* architecture, proposed analyses model and basic implementation aspects. The details are described in paper [17].

Architecture. The *Analyzer* allows to work with multiple opened projects at once, each representing one analytical research intent. Thus, we can divide the framework architecture into two separate levels, as it is depicted in Figure 1. The first one contains components, which are shared by all these projects. The second one represents components exclusively used and created in each opened project separately (repositories, storages, crawlers and entity managers).

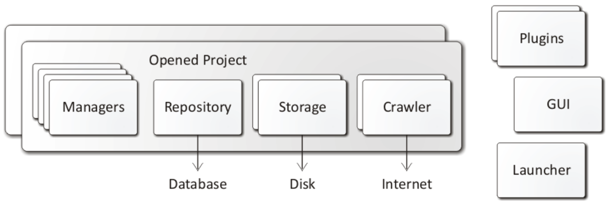

Fig. 1. Analyzer Framework Architecture

Repositories serve for storing all computed analytical data and the majority of project configuration metadata. Storages are used for storing document contents, i.e. binary contents of analyzed files. Finally, documents to be analyzed can be inserted into existing projects through import sessions (locally accessible files) or download sessions (downloading files from the Internet via crawlers).

The project layer contains a set of managers, which are responsible for creating, editing and processing of all analysis entities such as documents, collections of documents or reports over collections. As all computed analytical data are stored permanently in a repository, in order to increase efficiency, these managers are able to cache loaded data and some of them even to postpone and aggregate required update operations without violating the consistency.

Analyses. Although the framework enables also more complex usages, the standard life cycle of each project can be represented by the following phases.

1. Creation of a new project and configuration of its components,
2. Selection and configuration of analyses using available plugins,
3. Insertion of documents to be analyzed through import or download sessions,
4. Computation of analytical results over documents of a given relative age,
5. Selection and configuration of collections and clusters of them,
6. Document classification and assignment into collections, and
7. Computation of final statistical reports over particular collections.

Plugins. *Analyzer* itself provides a general environment for performing analyses over documents and collections of documents, but the actual analytical logic is not a part of it. All analytical computations and mechanisms are implemented in plugins. Not only that each particular plugin is intended only for processing of selected document types only, the user is also able to configure available plugins and, thus, adjust their behaviour to desired analytical intents.

The plugin functionality itself is provided through implemented methods, which are of eight predefined types listed in the following enumeration.

- The *detector* recognizes types of a processed document,
- The *tracer* looks for outgoing links in a given document,
- The *corrector* attempts to repair a content of a given document,
- The *analyzer* produces results over a given document,
- The *collector* classifies documents into collections of a given cluster,
- The *provider* creates reports over documents in a collection,
- The *viewer* serves for browsing computed results over a document, and
- The *performer* serves for browsing computed reports over a collection.

4 Processing of Incorrect Data

During the phase of document processing and result generating in *Analyzer* framework, *corrector* methods of available plugins are able to modify data contents of such documents. This feature is motivated primarily by the possibility of working with potentially incorrect documents.

In this section, we particularly focus on the problem of structural invalidity of XML documents. In other words, we assume the inspected documents are well-formed and constitute trees, however, these trees do not conform to a schema in

DTD [5] or XML Schema [9]. Having a potentially invalid XML document, we process it from its root node towards leaves and propose minimal corrections of elements in order to achieve a valid document close to the original one.

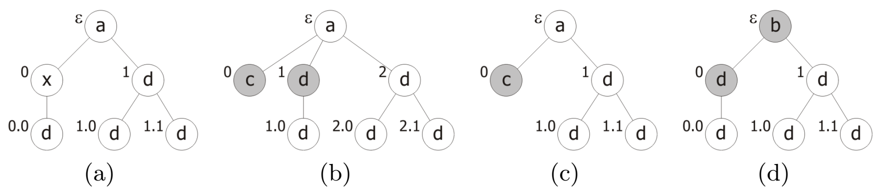

Fig. 2. Sample invalid XML tree with three possible minimal repairs

In Figure 2 we can see a sample correction process. Item (a) represents an original XML document, where element names are depicted by labels inside nodes. Without any detailed schema knowledge, assume only that element x at position 0 is not allowed. Processing this invalid tree, our algorithm finds three different minimal repairs, all of which are outlined in Items (b), (c) and (d).

The remaining parts of this section will present basic ideas of our correction model and proposed algorithms for finding structural repairs of invalid XML documents. Details of this proposal are presented in [16, 15].

4.1 Proposed Solution

Our correction framework is capable to generate local structural repairs for invalid elements. These repairs are motivated by the classic Levenshtein metric for strings. For each node in a given XML tree and its sequence of child nodes we attempt to efficiently inspect new sequences that are allowed by the corresponding content model and that can be derived using the extended concept of measuring distances between strings. However, in our case we do not handle only ordinary strings, but sequences derived from node labels with nested subtrees.

The correction algorithm starts processing at the root node and recursively moves towards leaf nodes. We assume that we have the complete data tree loaded into the system memory and, therefore, we have a direct access to all its parts. Under all conditions the algorithm is able to find all minimal repairs, i.e. repairs with the minimal distance to the grammar and the original data tree according to the introduced cost function.

Edit Operations. Edit operations are elementary transformations that are used for altering invalid data trees into valid ones. They behave as functions, performing small local modifications with a provided data tree. Despite the correction algorithm does not directly generate sequences of these edit operations, we can, in the end, acquire them using a translation of generated repairs, as it will be explained later. We have proposed and implemented edit operations capable to insert a new leaf node, delete an existing one and rename a label of a node.

Edit operations can be composed together into sequences. And if these sequences fulfil certain qualities, they can be classified as update operations. In this way we can work with update operations capable to insert a new subtree, delete an existing subtree and recursively repair a subtree with an option of changing a label of its root node.

Repairing Instructions. Assume that we are in a particular node in a data tree and our goal is to locally correct this node by correcting the sequence of its child nodes. Since the introduced model for measuring distances uses only non-negative values for the cost function, in order to acquire the global optimum, we can simply find minimal combinations of local optimums, i.e. minimal repairs for all subtrees of original child nodes of the inspected one.

However, we need to find all minimal repairs and even represent them in a compact repair structure. For this purpose we use repairing instructions. We have exactly one instruction for each edit operation and these instructions represent the same transformation ideas, however, do not include particular positions to be applied on. Having a sequence of instructions at a given level, we can easily translate it into all corresponding sequences of edit operations later on.

Correction Intents. Being in a particular node and repairing its sequence of child nodes, the correction algorithm generally has many ways to achieve the local validity proposing repairs for all involved child nodes. As already outlined, these actions follow the model of measuring distances between ordinary strings. The Levenshtein metric is defined as the minimal number of required elementary operations to transform one string into another.

We follow the same model, however, we have edit and update operations respectively and sequences of nodes. For example, an insertion of a new subtree at a given position stands for the insertion of its label into the corresponding node sequence and, of course, recursive generation of such new subtree.

The algorithm attempts to examine all suitable new words that are in the language of the provided regular expression restraining the content model of the inspected parent node. We do not generate word by word, but we inspect all suitable words statically using a notion of a correction multigraph. Correction intents represent assignments for these multigraphs, i.e. the recursive data tree processing in a top-down manner.

Correction Multigraphs. All existing correction intents in a context of a given parent node can be modelled using a multigraph for this node. Vertices of a multigraph for n child nodes can be divided into $n+1$ disjoint strata, vertices of each stratum correspond to states of the Glushkov automaton for recognising the provided regular expression. Edges in a multigraph are derived from the automaton transition function and they represent nested correction intents.

In order to find best repairs for a provided sequence of nodes, we need to find all shortest paths in the multigraph. Therefore, we first need all its edges to be associated with already evaluated nested repair structures and their minimal costs. And this represents nontrivial nested recursive computations. Anyway, we require that each edge can be evaluated in a finite time, otherwise we would obviously not be able to find required shortest paths at all.

Repairs Construction. Each correction intent can essentially be viewed as an assignment to the nested recursive processing. The correction of a provided data tree is initiated as a special starting correction intent for the root node and processing of every intent always involves the construction of at least the required part of the introduced multigraph with other nested intents.

Therefore, we continuously invoke recursive computations of nested intents. When we reach the bottom of the recursion, we start backtracking, i.e. encapsulating all found shortest paths into a form of a compact repair structure and, then, passing it one level up, towards the starting correction intent.

Having constructed a repair structure for the starting intent, we have found corrections for the entire data tree. Each intent repair contains encoded shortest paths and related repairing instructions. Now we need to generate all particular sequences of repairing instructions and translate them into standard sequences of edit operations. Having one such edit sequence, we can apply it on the original data tree and we obtain its valid correction with a minimal distance.

Correction Algorithms. Now, we have completely outlined the proposed correction model. However, there are several related efficiency problems that would cause significantly slow behaviour, if we would strictly follow this model. Therefore, we have introduced two particular correction algorithms, which are described in detail in [16]. They both produce the same repairs, but there are differences in their efficiency.

The first algorithm is able to directly search for shortest paths inside each intent computation and, therefore, does not need the entire multigraphs to be constructed. The next improvement is based on caching already computed repairs using signatures distinguishing different correction intents, but intents with the same resulting repair structure. This causes that this algorithm never computes the same repair twice. The second algorithm is able to compute lazily even to the depth of the recursion. We have achieved this behaviour by scattering all nested intents invocation and multigraph edges evaluation into small tasks, which are incrementally executed by a simple scheduler.

5 Query Analysis

In this section, we describe the second unique feature of *Analyzer* – *XQAnalyzer* – a tool designed to support studies that include analysis of a collection of XQuery programs. *XQAnalyzer* consumes a set of XQuery programs, converts them into a kind of intermediate code, and stores this internal representation in a repository. Subsequently, various analytical queries may be placed on the repository to determine the presence or frequency of various language constructs in the collection, including complex queries focused on particular combinations of constructs or classes of constructs.

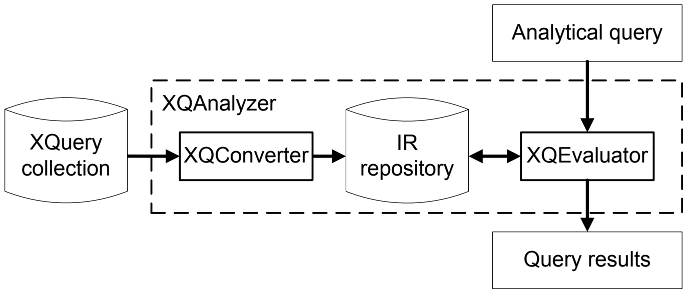

Fig. 3. The architecture of the *XQAnalyzer*

The architecture of the *XQAnalyzer* is shown in Figure 3. Each document from a given collection of XQuery programs is parsed and converted to the internal representation by the *XQConverter* component. The *XQEvaluator* component evaluates analytical queries and returns statistical results.

5.1 Analytical Queries

In the *XQAnalyzer*, the term *analytical query* denotes a pattern or condition placed on a XQuery program, usually a search for a feature. Each XQuery program in the collection is evaluated independently, producing either a boolean value or a hit count representing the presence or number of occurrences in the program, respectively. The *XQEvaluator* then returns various statistical results like the percentage of programs which contain the searched feature or the histogram of hit counts over the repository.

Given the fact that the tool is designed for research in the area of XML and, in particular, XQuery, the best choice would be a query language derived from XPath. XPath is naturally well-known in the community and it is designed to place pattern-like queries on tree structures – in our case, a tree is a typical representation of a program during early stages of its analysis.

5.2 Internal Representation of XQuery Programs

The key issue in the design of *XQAnalyzer* is the internal representation of XQuery programs. In our approach, we do not want to limit the nature of the analytical queries; therefore, the internal representation must store any XQuery program without loss of any feature (perhaps except of comments). Furthermore, the internal representation is exposed to the user via the query interface; therefore, it should be as simple as possible. Finally, the internal representation affects the performance of the *XQEvaluator*.

The W3C standards related to XQuery define at least the following two formalisms that might be used as a base for our internal representation:

- The *XQuery Grammar* (in Extended Backus-Naur Form) defined in [4].
- The *Normalized XQuery Core Grammar* (also in EBNF) defined in [8].

Note that the XQuery formal semantics [8] is defined in terms of static/dynamic evaluation rules that may be considered a kind of internal representation too. However, their application in our analytical environment would be impractically difficult.

Among the existing formalisms mentioned so far, we have chosen the Normalized XQuery Core Grammar. There are the following reasons behind this decision:

- It is a part of the standard, therefore well known and not skewed towards any evaluation strategy.
- It is smaller than the full XQuery Grammar and it hides the redundant features of the XQuery language.

The final set of nonterminals is listed in Tab. 1 together with their frequency in selected collections of XQuery programs (see Sec. 5.3). When our internal representation is presented in the form of a XML document, these nonterminals become *XML elements.*

The rest of the semantic information is enclosed in *XML attributes* attached to these elements. These attributes contain either data extracted from the source text (like names of variables or contents of literals) or additional semantic information (like the axis used in an XPath axis step). In addition to these data required to preserve the semantics, we also added attributes that may help recovering the original syntax before the normalization to XQuery Core (e.g. whether the abbreviated or the full syntax was used in axis step).

5.3 Results

Since there is no standardized collection of real-life XQuery programs yet (except of small benchmarks like XMark [1]), we have chosen two artificial collections associated to the W3C XQuery language specification: The XQuery Use Cases [6] and the XQuery Test Suite [19]. The Use Cases collection consists of 85 "text-book" XQuery programs prepared to demonstrate the most important features of the language, the Test Suite collection contains 14869 small XQuery programs created to cover all features (the remaining 252 files in the original collection contain intentional parse errors). Although the Test Suite collection is more than 100 times larger in terms of the number of files, the real ratio of sizes (in terms of the number of AST nodes) is 31:1 because the Use Cases files are larger.

In Tab. 1 we show the frequency of core elements of the language, named accordingly to the abstract grammar nonterminals derived from the Normalized

XQuery Core Grammar (see Sec. 5.2). The percentages are defined by the number of occurences divided by the total number of abstract syntax tree nodes in the collection (which was 4 469 for the Use Cases and 138 949 for the Test Suite).

Besides the obvious difference between the two collections, corresponding to their purpose, there are the following noticeable observations: The frequency of quantified expressions (`some` or `every`) is about eight times smaller than the frequency of `for`-expression. The `if`-expression is quite rare – once per 30 `for`-expressions or 50 operators. A number of features like `ordered`/`unordered`-expressions are omitted in the Use Cases. While frequent in the Test Suite, the comma operator is surprisingly rare in the Use Cases.

Table 2 shows the use of the twelve XPath axes. The percentages represent the frequency of individual axes among all axis step operators in the collection (which was 638 for the Use Cases and 6 623 for the Test Suite). Notice that the results correspond to the traditional belief that many axes are extremely rare.

6 Conclusion

The main aim of this paper was to describe several research problems related to a complex extensible framework for analyses of real-world XML data called *Analyzer*. Firstly, we have proposed a correction framework dealing with invalid nesting of elements in XML documents using the top-down recursive processing of potentially invalid data trees. Contrary to existing approaches, we have considered the class of single type tree grammars instead only local tree grammars. We are able to find all minimal repairs. Secondly, we described *XQAnalyzer* and implemented a tool for analysis of collections of XQuery programs. *XQAnalyzer* works with a set of XQuery programs and translates them into an intermediate code. Subsequently, analytical queries may be placed over these translations to get the precence of quantity of specific constructs.

In our future plans, we will focus on further improvements of existing plugins related to XML data analyses and their exploitation in throughout analysis of both current state of real-world XML documents and evolution of XML data in the following months. We plan to repeat the analysis monthly and publish the new as well as aggregated results on the Web. We believe that such a unique analysis will provide the research community with important results useful for both optimization purposes as well as development of brand new approaches. Concurrently, we will shift our target area to the new types of data such as RDF triples, linked data, ontologies etc.

Acknowledgement

This work was partially supported by the Czech Science Foundation (GAČR), grants number 201/09/P364 and P202/10/0573.

Element	Use Cases	Test Suite
AtomicType	0.27%	2.49%
Axis	14.28%	4.79%
BaseURIDecl	—	0.04%
BindingSequence	3.62%	1.11%
BoundarySpaceDecl	—	0.07%
CData	—	0.01%
CaseClauses	0.02%	0.03%
CharRef	—	0.02%
CommaOperator	0.04%	2.04%
ConstructionDecl	—	0.04%
Constructor	3.36%	2.07%
Content	4.03%	2.11%
ContextItem	0.22%	0.11%
CopyNamespacesDecl	—	0.02%
DefaultCase	0.02%	0.03%
DefaultCollationDecl	—	0.01%
DefaultNamespaceDecl	—	0.12%
ElseExpression	0.07%	0.08%
EmptyOrderDecl	—	0.03%
EmptySequence	0.02%	0.63%
EntityRef	—	0.01%
Extension	—	0.03%
FLWOR	1.97%	0.79%
ForClause	2.10%	0.59%
FunctionBody	0.40%	0.16%
FunctionCall	5.77%	17.11%
FunctionDecl	0.40%	0.16%
Hint	0.38%	0.01%
IfExpr	0.07%	0.08%
InClauses	0.27%	0.15%
KindTest	4.83%	0.80%
LetClause	1.25%	0.32%
Literal	4.61%	20.32%
ModuleDecl	0.07%	0.00%
ModuleImport	0.07%	0.03%
Name	4.21%	2.14%
NameTest	10.14%	4.08%
NamespaceDecl	0.20%	0.18%
OperandExpression	0.02%	0.03%
Operator	3.85%	8.43%
OptionDecl	—	0.01%
OrderedExpr	—	0.01%
OrderingModeDecl	—	0.02%
Path	10.02%	2.51%
PragmaList	—	0.03%
QuantifiedExpr	0.27%	0.15%
QueryBody	1.83%	10.70%
ReturnClause	2.04%	0.90%
SchemaImport	0.38%	0.17%
String	6.82%	2.12%
TestExpression	0.34%	0.23%
ThenExpression	0.07%	0.08%
TupleStream	1.97%	0.79%
Type	0.98%	2.62%
Typeswitch	0.02%	0.03%
UnorderedExpr	—	0.01%
ValidateExpr	—	0.02%
VarDecl	—	2.42%
VarRef	8.68%	3.47%
VarValue	—	2.42%

Table 1. The elements of the internal representation

Element	Use Cases	Test Suite
child	71.63%	82.67%
descendant	—	0.21%
attribute	5.33%	3.70%
self	—	0.36%
descendant-or-self	23.04%	10.40%
following-sibling	—	—
following	—	0.44%
parent	—	0.50%
ancestor	—	0.44%
preceding-sibling	—	0.42%
preceding	—	0.42%
ancestor-or-self	—	0.44%

Table 2. Axis usage

References

1. Afanasiev, L., Marx, M.: An analysis of xquery benchmarks. Inf. Syst. 33(2), 155–181 (2008)
2. B. Bouchou, A. Cheriat, M. H. Ferrari Alves, A. Savary: Integrating Correction into Incremental Validation. In: BDA (2006)
3. Bex, G.J., Neven, F., den Bussche, J.V.: Dtds versus xml schema: a practical study. In: WebDB '04. pp. 79–84. ACM, New York, NY, USA (2004)
4. Boag, S., Chamberlin, D., Fernndez, M.F., Florescu, D., Robie, J., Simon, J.: XQuery 1.0: An XML Query Language (Second Edition). W3C (December 2010), http://www.w3.org/TR/xquery/
5. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C (November 2008), http://www.w3.org/TR/xml/
6. Chamberlin, D., Fankhauser, P., Florescu, D., Marchiori, M., Robie, J.: XML Query Use Cases. W3C (March 2007)
7. Corrector Prototype Implementation, http://www.ksi.mff.cuni.cz/~svoboda/
8. Draper, D., Fankhauser, P., Fernández, M., Malhotra, A., Rose, K., Rys, M., Siméon, J., Wadler, P.: XQuery 1.0 and XPath 2.0 Formal Semantics. W3C (January 2007)
9. Gao, S., Sperberg-McQueen, C.M., Thompson, H.S.: W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. W3C (December 2009), http://www.w3.org/TR/xmlschema11-1/
10. Mignet, L., Barbosa, D., Veltri, P.: The xml web: a first study. In: Proceedings of the 12th international conference on World Wide Web. pp. 500–510. WWW '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/775152.775223
11. Mlynkova, I., Toman, K., Pokorny, J.: Statistical analysis of real xml data collections. In: COMAD'06. pp. 20–31. Tata McGraw-Hill Publishing, New Delhi, India (2006)
12. S. Flesca, F. Furfaro, S. Greco, E. Zumpano: Querying and Repairing Inconsistent XML Data. In: WISE '05. LNCS, vol. 3806/2005, pp. 175–188. Springer (2005)
13. S. Staworko, J. Chomicky: Validity-Sensitive Querying of XML Databases. In: Current Trends in Database Technology EDBT 2006, DataX06. Lecture Notes in Computer Science, vol. 4254/2006, pp. 164–177. Springer (2006)
14. Stárka, J., Svoboda, M., Sochna, J., Schejbal, J.: Analyzer 1.0. http://analyzer.kenai.com/
15. Svoboda, M.: Processing of Incorrect XML Data. Master Thesis, Charles University in Prague, Czech Republic (September 2010), http://www.ksi.mff.cuni.cz/~mlynkova/dp/Svoboda.pdf
16. Svoboda, M., Mlýnkova, I.: Correction of Invalid XML Documents with Respect to Single Type Tree Grammars. In: NDT 2011. Communications in Computer and Information Science, vol. 136. Springer, Macau, China (2011), [to be published]
17. Svoboda, M., Stárka, J., Sochna, J., Schejbal, J., Mlýnkova, I.: Analyzer: A framework for file analysis. In: BenchmarX '10. pp. 227–238. Springer-Verlag, Tsukuba, Japan (2010), http://www.springerlink.com/content/078819t6645j6268/
18. U. Boobna, M. de Rougemont: Correctors for XML Data. In: Database and XML Technologies. LNCS, vol. 3186/2004, pp. 69–96. Springer (2004)
19. W3C: XML Query Test Suite (November 2006)
20. Z. Tan, Z. Zhang, W. Wang, B. Shi: Computing Repairs for Inconsistent XML Document Using Chase. In: Advances in Data and Web Management. LNCS, vol. 4505/2007, pp. 293–304. Springer (2007)