Charles University in Prague

Faculty of Mathematics and Physics

DOCTORAL THESIS

Jakub Stárka

Analyses of Real-World Data and Their Exploitation

Department of Software Engineering

Supervisor of the doctoral thesis: RNDr. Irena Holubová, Ph.D.

Study programme: Software Systems

Prague 2013

Let me thank Bára, my parents, friends and colleagues from the XML and Web Engineering Research Group for supporting me throughout my lengthy studies.

Let me also thank my supervisor RNDr. Irena Holubová, Ph.D. for her exemplary leadership and direction and all the anonymous reviewers of my papers for helpful suggestions and insightful opinions.

Finally, let me thank the Czech Science Foundation (GAČR) and the Grant Agency of Charles University (GAUK) for supporting me through their grants GAČR P202/10/0573, GAČR P201/09/0990, GAČR P201/09/P364, GAUK 410511, GAUK SVV-2010-261312, GAUK SVV-2011-263312, GAUK SVV-2012-265312 and GAUK SVV-2013-267312, respectively.

Název práce: Analýzy reálných dat a jejich využití

Autor: Jakub Stárka

Katedra: Katedra softwarového inženýrství

Vedoucí disertační práce: RNDr. Irena Holubová, Ph.D.

Abstrakt: Znalost reálných dat je základem pro optimalizaci mnoha technik zpracování dat. Jejich získání, analýza či integrace zahrnují mnoho problémů, na které je zaměřena tato práce. Mezi tyto hlavní problémy patří např. automatické stahování dokumentů, extrakce dat a jejich analýza, či odvozování schémat.

V této práci popíšeme komplexní framework, který umožňuje opakovaně provádět statistickou analýzu nad reálnými XML dokumenty, které jsou získané z internetu. Také navrhneme několik charakteristik pro XML dokumenty, RDF trojice a XQuery dotazy včetně podrobných výstupů analýz nad několika veřejně dostupnými kolekcemi dat. V neposlední řadě popíšeme rozšiřitelný nástroj pro odvozování XML schémat. Díky jeho modulárnímu designu je možné kombinovat několik nezávislých přístupů pro jednotlivé kroky. V rámci práce nepopíšeme jen samotný framework, ale i oblast odvozování jako takovou a s ní související problémy.

Klíčová slova: analýza dat, extrakce dat, odvozování schémat

Title: Analyses of Real-World Data and Their Exploitation

Author: Mgr. Jakub Stárka

Department: Department of Software Engineering

Supervisor: RNDr. Irena Holubová, Ph.D.

Abstract: The typical optimization strategy of many data processing techniques is exploitation of the knowledge of constructs typically used in real-world applications. However, such approach requires a repeatable, updatable and detailed analysis of a representative data set. Having such a requirement a number of related problems arises, such as automatic crawling of the data, data extraction, schema inference, and efficient performance of analyses over a huge data volume as well as exploitation of the results in current applications.

In this thesis we describe a complex framework for performing statistical analyses of real-world documents and we propose characteristics that appropriately capture and describe features of XML documents, RDF triples and XQuery queries. Additionally we provide experimental results over a few selected real-world data sets. Last but not least we introduce an easily extensible tool that enables one to implement, test and compare new modules of the XML schema inference process. We describe not only the framework, but the area of schema inference in general, including related work and open problems.

Keywords: data analysis, data extraction, schema inference

Contents

List of Figures

List of Tables

1. Introduction

Currently, the World Wide Web is used as a primary source of information and users are sending billions of emails and publishing millions of documents, images and video clips every day. Moreover, e-shops or online stock exchanges make the Internet one of the most important business platforms. Thus the amount of data interchanged between users and corporations is huge and the requirements for fast and efficient processing and managing of documents rise. Data generated by companies are an important source of information for future decisions and their analyses can be used to reveal wrong steps in the past, to predict new trends in the future, or to attract new customers. Not surprisingly, the data become a very expensive commodity and the tools for their acquisition, manipulation and processing are intensively studied and demanded.

As the need for data increases, the performance requirements for the tools and databases are higher, so more efficient algorithms are needed. Consequently, there arises a necessity for examination of common characteristics of a data. For instance, although XML [1], as a main format for data exchange, offers many constructs for data representation, only a small subset of them are used in real-world data [2, 3, 4, 5]. The information about the structure of the most common archetypes can be used to propose more efficient algorithms and data structures for their storage and efficient querying. For example, the finding that the depth of XML documents is on average less than 10 [6] is widely exploited in techniques which represent XML documents as a set of points in a multidimensional space and store them in corresponding structures, e.g., R-trees, UB-trees, or BUB-trees [7]. Likewise, similarity of XML schemas is currently exploited in many approaches, typically as a kind of optimization heuristics. The current approaches [8, 9, 10] are based on the idea of exploitation of various *matchers*, i.e., functions which evaluate similarity of selected simple data characteristics (e.g., similarity of element names, similarity of number of subelements etc.). Their results are then aggregated to a resulting *composite similarity measure* using a kind of weighted sum. The problem is how to set the weights so that the result reflects the reality. And the solution can be found in statistical analyses of real-world data.

Another research area which widely exploits the knowledge of complexity of real-world data is inference of XML schemas from a given sample set of XML documents. Since according to Gold's theorem [11] regular languages (i.e., those generated by XML schemas) are not identifiable only from positive examples (i.e., sample XML documents which should be valid against the resulting schema), the existing methods need to exploit either heuristics or a restriction to an *identifiable* subclass of regular languages. The question is which subclass should be considered. The authors of papers [12, 13] result from their analysis of real-world XML schemas [5] and define the classes so that they cover most of the real-world examples. In other words, the identifiable subclass corresponds to a realistic situation. These statistics can be also useful for data format specification, i.e., the new specification is based on the usage of the most common patterns used over the Internet and their analysis.

On the other hand, despite the importance of the schema the analyses of real-world XML data show that a significant portion of XML documents [6, 14] still have no schema at all. Hence, approaches which focus on automatic inference of schema based on a set of XML documents can be used for efficient integration of these documents into an existing system [15], for improvement of compression ratio [16, 17], or it can be used for further communication with other services.

Considering the problem of structural analysis in more detail, numerous problems arise when we want to analyze the data. In this case, we have to answer questions like: “How can we get the data?”, “Are the data correct?”, or “Are the data complete?”. For example, the government of the Czech Republic publishes data about public contracts with a price higher than a limit provided by the law, but contracts with a lower price can be found at the Web sites of the cities, regions, etc. These data contain only the identification number of suppliers so it has to be connected to a diverse data set from other publishers. Some countries continue in the trend of publishing governmental data as open data[1], so the data are directly enriched by other available data sets, i.e., the data are combined, corrected, or additional information is computed and added to the data set. Unfortunately, most of the published data are still published in a form of HTML [18] tables, spreadsheets, or as scanned images which leads to the demand for automatic extraction tools that are able to extract, clean and enrich the resulting data sets automatically. Thus we can differentiate the availability of data as follows:

- data are available in a structured format (e.g., RDF [19], XML, relational databases) and we have a full access to them,
- data are available in a structured format, but the source is distributed so we have to query different sources, download dumps from various locations or use Web Services [20] to get them,
- data are available in a semi-structured format (e.g., HTML or structured text) and additional processing is necessary to get the data,
- data are unavailable or hidden in proprietary formats with low chances to extract valuable fields.

Obviously the first option (i.e., work with structured data like XML) does not need any preprocessing before we analyze the data (possibly we can validate or correct the data if they are not from a trustworthy source). However, the others need at least data acquisition, correction, and in the latter case data extraction, before we can start with their analysis.

Based on the previous observations we can divide the main requirements for data processing as follows:

R1 Document acquisition – The easiest situation (but usually uncommon) is to own the data in a structured form. It means we have the access to the database or (e.g., XML) files which gives us a simple possibility of data analysis. Despite having all data, we might still need to get additional data from external sources, i.e., to download referred documents, to connect to third party systems, etc.

R2 Data extraction – Once we have an access to the data, but not in a structured form (e.g., simple text documents), a data extraction tool is necessary to get the hidden information or included links. Although the documents are unstructured they still can be used as a valuable source of information and we can use user-defined templates to extract data from semi-structured documents. For example, official letters have typically a heading, a date, an inside address, a greeting, a

[1] `http://opendatahandbook.org/en/`

subject, a body, and a signature. Or, contracts usually contain information about a contractor, a supplier, and a subject. Thus it is possible to specify the way how to extract data from different sources with different formats through various specialized data extractors (which can use regular expressions or format-specific selectors).

R3 Static analysis – In general, we want to know how the documents are structured and to use this information to improve the way we work with them, i.e., the integration into existing systems, their querying, batch processing or any other way to analyze the data. A static analysis means that the results are calculated without respect to the changing data, i.e., snapshot analysis.

R4 Dynamic analysis – On the other hand, a dynamic analysis means that the results are calculated repeatedly through the time which can be used to follow up trends and to quickly respond to changes.

R5 Data processing – Apart from statistical analysis, we can use the gathered data to generate new information. A typical example for XML documents is inferring a schema. For this reason we can use different formats and sources. We can work with XML documents to get a sample of data to generate an inferred general schema. Besides this, the schema inference process can be more efficient with addition of other types of documents, e.g., an old obsolete schema or commonly used queries.

1.1 Aims of This Thesis

Requirements R1 – R5 outline the general scope of this thesis. We want to design and implement a framework which is able to integrate set of tools to allow a user to extract, analyze, process, and visualize structured or semi-structured data from the Web. In the following paragraphs we describe the main aims of this thesis in more detail.

1.1.1 Document Acquisition and Analysis

Data (or document) analysis comprises many independent steps which have different requirements to the user, software and hardware. Namely, the user has to specify the analyzed data sets, (s)he must configure analytic tools or implement custom processing tools. For example, let us have a set of documents about public procurements published on the Web. Firstly, we have to implement a downloader to retrieve the documents from the Web. Then we have to check their correctness, e.g., whether all required fields are filled in. In the next step we implement an analytic method, e.g., evaluation of the average price for each month. Finally, we must create a chart to present our results. For these purposes many standalone tools are needed. Last but not least, the hardware has to be prepared to work with a huge amount of data, it has to be able to offer Internet connection (if the source data are gathered from the Internet), etc. For these reasons it is necessary to create a sequence of tools which are able to communicate mutually or, at least, to pass the data to the next tool in the chain. Another possibility is to offer one framework which performs all the processing steps and implements the main components of the system.

In this thesis, we introduce the architecture and functionality of *Analyzer* (see Section 2), a framework that enables automatic repeatable (i.e., dynamic) and extensible analyses of real-world data. It is a complex tool that supports not only the analytical part, but also various "supportive" functions, such as document crawling, data correction, or analytic visualizations.

With regard to the current support of XML, we provide results of statistical analyses of real-world XML data. We describe the common properties of XML documents from different sources, the commonly used axes in used XQuery [21] queries and we show the structure of real-world semantic data.

1.1.2 Data Extraction

Although *Analyzer* is a universal tool for data analysis, it was not designed to store metadata about documents or to enrich the data, i.e., to combine different sources to get a more complex data set. For example, we want to create data about the public procurements. These data are published over several sources and thus it is not possible to query the whole data set to get the aggregated data. So we need to create a single centralized database.

In this thesis we describe the way how the data can be extracted from different HTML sources. Since this topic is very broad, we focus on creation of template-driven (i.e., user-driven) application working with our designed scraping language (see Section 5).

1.1.3 Data Processing

There are many possible ways how to utilize the gathered (and eventually enriched) documents. In this thesis we show a few examples how the data can be extracted and utilized:

- Structural Analysis – Once the source of the data is selected, we have to decide what to do with the extracted documents or data fields. One possibility is their direct processing as far as we know that the data will not change or we will not add new data sets. Then we can run batch analytic computation to get structural or semantic information (see Sections 2.6 and 2.7).

- Data Analysis – Other possibility is to follow the semantic meaning of the data to get information about the data itself (see Section 3).

- Inference – Another utilization of the data comprises the possibility to infer a schema (or an ontology) to describe the data. This schema can be then used to validate new documents from the same domain or it can be used as a basis for information systems to provide specific Web Services. In this thesis we describe enrichment of commonly used inference methods. For this purpose we do not use in the inference process only XML documents, but we include other sources of information like XQuery queries or obsolete schemas (see Section 6).

1.2 Proposed Approach

From requirements R1 – R5, it is obvious that a complex framework or a set of tools which would automate the tasks done manually is the most suitable solution. Such a framework should provide a tool for Web crawling that is able to download documents recursively, to store the documents, to prepare them for an analysis, to make the computations, and to present the results in a human readable form. As we have already mentioned, such a framework is proposed in this thesis and it is called *Analyzer*.

An important part of *Analyzer* is the crawling module. Our first implementation downloaded documents from Web pages, i.e., the links were recognized as the content of attribute `href`. Soon, we realized that in many cases the documents are hidden in so-called *Deep Web*, in other words the documents are accessible via HTML forms or AJAX[2] calls. *Deep Web* represents a large fraction of the structured data on the Web [22], so it is important to crawl it effectively. Additionally, we realized that the crawled Web pages can contain different meta data (e.g., the last modification date, the author of the document, or various license information), which can be interesting for further analysis and data publication. Thus we decided to separate the crawling module into a standalone tool called *Strigil* which is able to download documents, browse the *Deep Web*, and extract specific data based on user-defined templates.

With the results of the *Analyzer* we confirmed the results of previous papers [14, 5, 6] that most of the XML documents do not contain an XML schema. Based on these observations we focused on a schema inference process and proposed a tool *jInfer* for schema inference.

1.2.1 Document Acquisition and Analysis

Currently there exist several papers that describe the results of an analysis of real-world XML data. Firstly, there occurred several analyses of the structure of DTDs [2, 3, 4] which analyzed mainly the complexity of content models and usage of various constructs. With the arrival of XML Schema [23] a natural question arisen: Which of the extra features of XML Schema not allowed in DTD [1] are used in practice? Paper [5] is trying to answer it using a statistical analysis. Finally, there exist also papers that analyze the structure of XML documents [24] regardless an eventually existing schema.

However, though the amount of existing works is significant and the findings are important, all the papers have a common disadvantage. Sooner or later each analysis becomes obsolete and it should be repeated. However, the respective crawlers, data analyzers and their settings are not available any more or they have limited functionality and cannot be extended with new features easily. From the problems outlined, it is obvious that a certain tool is needed to automate the tasks. Such a tool should ensure that the user will be provided with document acquisition, preprocessing, analysis and visualization. Such a tool would increase the efficiency with the data manipulation and it would allow the user to focus on metrics instead of common tasks. In other words, it provides all essential functionality for an easy management of files to be analyzed, configuration and execution of selected analyses and an advanced graphical user interface (GUI) for browsing the generated outputs.

[2]Asynchronous JavaScript calls to insert dynamic content of a Web page

Thus we designed and implemented a tool called *Analyzer*, which is described in Section 2. It is a complete framework for performing statistical analyses of real-world documents. It provides all essential functionality for an easy management of files to be analyzed, configuration and execution of selected analyses and an advanced graphical user interface for browsing generated reports. To ensure all the above indicated features in a user-friendly manner, the key advantage of *Analyzer* is extensibility. This not only means the ability to implement own and more suitable kernel components responsible, e.g., for storing computed analytical data, but primarily the open concept of plugins. *Analyzer* provides a general environment, whereas all analytical computations themselves are defined solely within the implementation of plugins. The user is therefore expected to first install *Analyzer* itself and then eventually create his/her own plugins designed to correspond to the determined research intents.

In particular, *Analyzer* is a standalone application consisting of:

1. a download engine for automated crawling of documents with link recognition features (typically in XML or HTML documents),
2. a preprocessing module which is able to correct incomplete or invalid documents,
3. a set of analytic modules to examine different document features,
4. an aggregation component that assigns documents into collections based on a common feature, and
5. a visualization module to display or export computed results in a form of charts or tables.

The overall picture of the analytic process is depicted in Figure 1.1.

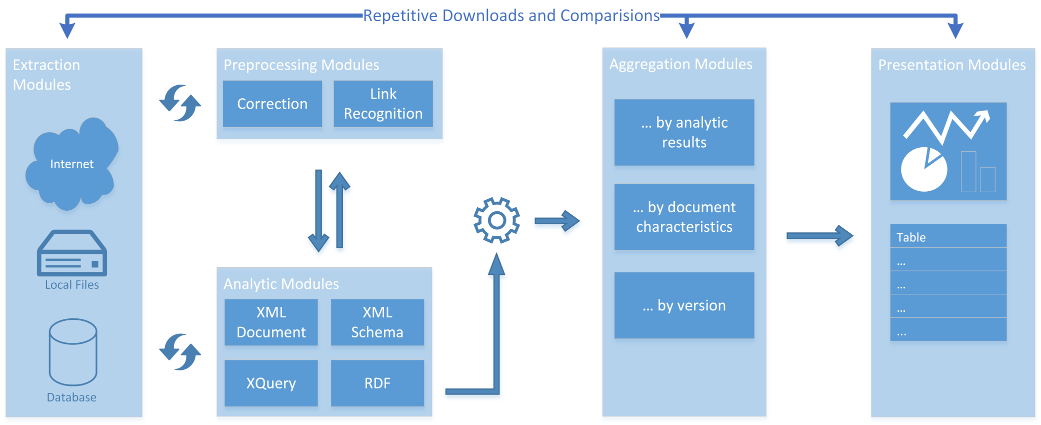


Figure 1.1: Analytic process of *Analyzer*

The subject of analyses are documents, which can be inserted into *Analyzer* in different ways. Firstly, it is possible to insert documents directly from a local file system. These documents can be previously manually downloaded by the user or documents generated by an information system. The second way is to download documents using one of the supported crawlers. In this thesis we only expect one simple crawler which is able to start at an *initial URL*, recognize links and download all documents up to a user-defined depth.

As depicted in Figure 1.1 the analytic process can contain preprocessing steps before the analyses are done. It is the only possibility when a document can be changed (although the original file is kept and available to the analytic methods if required). The reason is that the set of automatically or manually downloaded documents from the Internet can contain incomplete or corrupted documents [14]. So the aim of these preprocessing modules is to transform the documents into a form which the analytic methods are able to process. Obviously these methods do not have to be used and the analyses can be aimed, e.g., to the number of incorrect documents.

The next step is execution of analytic plugins. *Analyzer* offers a developer kit with Java interfaces to create new plugins or to extend the existing ones. The available plugins are listed and the user can select which should be used and what MIME[3] type of documents they should be applied to. Additionally the order of the selected plugins matters. For example, the user can first analyze the number of unfinished tags, then apply a correction plugin to create a valid XML document and, finally, compute the average depth of each document.

Once the analyses are computed, it is possible to create collections. A collection is a set of documents with a common feature (which is computed by an analysis). Over these collections we can run aggregation functions to get overall reports, e.g., the average depth of XML documents with regard to the document size. The results of the analytic methods and the reports can be visualized in a form of charts or tables (*Analyzer* supports visualization as a table, an HTML page or a chart with usage of *JFreeChart* library[4]). A sample screen shot image is provided in Figure 1.2.

Additionally *Analyzer* supports multiple versions of the same document. It is necessary if we want to analyze the data characteristics through the time which fulfils requirement R4, i.e., the dynamic analysis.

In the following paragraphs we describe results of custom modules implemented as a part of *Analyzer* and outlined in Figure 1.3.

XML Analyses As a preliminary proof of the concept we prepared and analyzed a sample of XML data from publicly available sources (e.g., the U.S. federal executive branch data sets[5], the Open Data Euskadi[6], or the U.S. congress[7]). We also used the *OpenTravel* specification[8] as a sample of XML Schema documents (XSDs) and compared their versions over last 9 years. We implemented several plugins to cover basic characteristics of XML, DTD and XML Schema based on previously published statistics [3, 25], such as:

- XML documents – maximum depth of the document, complexity of recursion of elements, maximum and average fan-out, usage of XML Schema versus DTD, etc.

- DTD documents – number of declarations of elements or attributes, used content types, attribute optionality, or usage of keys, etc.

[3]Multipurpose Internet Mail Extensions (MIME) is an Internet standard often used to describe content type.

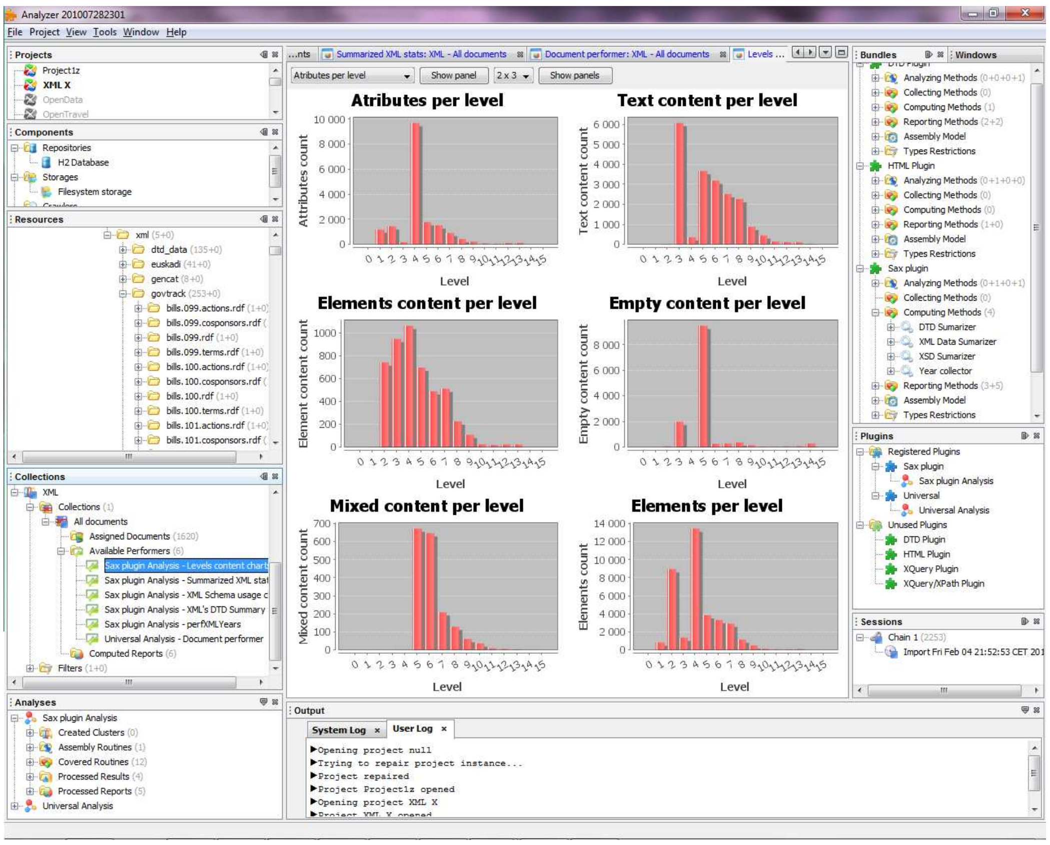
[4]`http://www.jfree.org/jfreechart/`

[5]`http://data.gov`

[6]`http://opendata.euskadi.net/`

[7]`http://www.govtrack.us/data/rdf/`

[8]`http://opentravel.org/`



```
▶Opening project null
▶Trying to repair project instance...
▶Project repaired
▶Project Project1z opened
▶Opening project XML X
▶Project XML X opened
```

Figure 1.2: *Analyzer* – a screen shot of the application

- XML Schema documents – the usage of restrictions and extensions, type specifications, etc.

According to the results we can in general say that a typical XML document is shallow which corresponds to the previously published results [14]. Additionally, we can see that, quite naturally, the XML schemas are getting more complex and their depth is increasing. On the other hand, the usage of XML Schema constructs is stagnating. The complete characteristics and especially all interesting results can be found in Section 2.6.

XQuery Analyses As we have mentioned, there exist several papers dealing with analyses of XML data. However, currently there exists no paper that focuses on XML data operations, namely XML queries. The most common tools, i.e., XQuery 1.0 [21] and XSLT 2.0 [26] are powerful, Turing-complete [27] languages; however, their applications usually solve relatively simple problems like generating HTML pages or transforming XML between two schemas. Consequently, it is often believed that most applications use only small subsets of these languages. This observation is also supported by the fact that the most popular textbooks on XQuery or XSLT do not cover the languages exhaustively.

From the perspective of the implementor of an XQuery or XSLT processor, this observation suggests that a number of language features is rarely used and, therefore, not worthy of aggressive optimization. For example, the *following*/*preceding* axes [28] are used significantly less frequently than the *child*/*descendant* axes; consequently, the

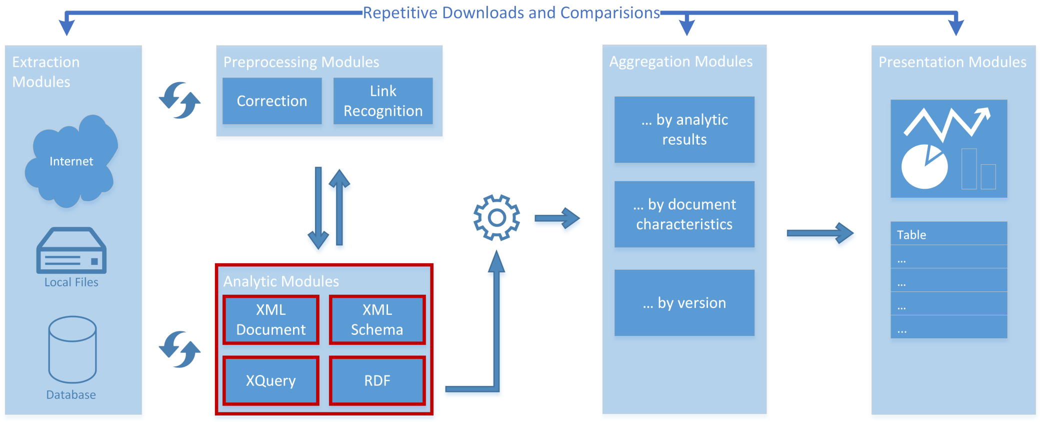


Figure 1.3: *Analyzer* – custom modules

majority of indexing and querying techniques like twig joins [29] are limited to the *child*/*descendant* axes.

Note that we introduced the observation with the clause "it is often believed"; indeed, it was probably never confirmed by any statistically significant study. Thus we decided to design and implement a new module for *Analyzer* which is able to analyze the queries.

A typical approach of researchers analyzing programming languages is to implement a simple tool for recognizing words in the target language, summarize these words and so get the frequency of their usage. Due to extreme *context dependency* of XQuery, such simple tools cannot work correctly and they often lead to wrong results. For example, a short and syntactically correct XQuery program

```
for $for as for in "for" return <for/>
```

contains the word "`for`" five times, each time in a different meaning depending on its position in the program. The researcher with a simple tool that is probably looking for the word "`for`" in the meaning of a *for*-clause from the *FLWOR* expression gets misleading results. That is why a more precise tool needs to be implemented, at least as strong as a correct *lexical scanner* of XQuery, which can also recognize the meaning of the scanned words. This conclusion leads us to an idea of *general observation*, i.e., creating a data structure representing the input program together with a meta-language for querying the structure. A simple and natural way we utilize in this thesis which enables one to reach this target is to build a *syntax tree*. It can be then easily transformed into XML representation and queried using, e.g., XPath [28].

The process of building the internal representation and its querying is described in Section 4. In Section 2.7.3, we present the analyzed frequency of core elements of the language associated to the W3C XQuery language specification (the *XML Query Use Cases* [30] and the *XML Query Test Suite* [31]). Note that the axis usage results correspond to the traditional belief that many axes are extremely rare.

RDF Triples Analyses Last but not least, to demonstrate the versatility of *Analyzer* we focused on analysis of highly linked data sets from available dumps of the Linked

Data Cloud[9] (e.g., the ACM proceedings[10], the English *DBPedia*[11], etc.). In this part we designed own metrics for description of the structure and depth of linkage to be used in the optimization process – issues previously identified as open problems of the existing approaches from the area of RDF triples storing, indexing and querying [32]. We propose a set of characteristics of RDF triples and provide experimental results over several selected data sets in Section 3.

First of all, the majority of indexing approaches (e.g., the Hexastore [33] or the BitMat Index [34]) proposes to store components of RDF triples and triples themselves separately (even using fairly different structures) in order to reduce space requirements. Knowledge of string features of these component values could support this practice.

The second group of characteristics worth studying is related to query evaluation and, in particular, access patterns to individual triple components. In case of full-text querying, we usually do not care which particular triple component should match the queried value, but in case of structural querying like SPARQL [35], we need to have suitable indices allowing us to efficiently access particular components according to the prompted query. These indices can be built, for example, on nested lists (Hexastore [33]) or B^+-trees (RDF-3X [36]).

Finally, we can even attempt to study more complex characteristics based on the structure of RDF graphs. When using SPARQL with queries based on graph patterns, we often need to do operations similar to traditional joining in relational databases, only with the difference that we are working with RDF triples, i.e., graph data. This joining can be supported by appropriate indices as well. Like, for example, pre-computed paths (RDF-3X [36]) or stars (Structure Index [37]).

Paper [38] describes the analysis of more than 1.5 million FOAF[12] documents. In particular, they inspected the usage of the FOAF namespace, host names and particular properties, as well as the relationships of a person in a group and other components of a social network. In general, this work describes several interesting characteristics, but its impact and context is very restrained. Similarly, the general statistics of the Linked Data cloud are described in [39]. The authors aimed at characteristics and link statistics between selected data sets. These data sets were divided by different thematic domains, for which several ingoing and outgoing statistics were computed. Provenance, licensing and data set-level metadata published together with these data sets were also considered. According to the discussed motivation, we proposed several characteristics that may be useful to know about RDF data we want to store, query or process in a different way. First of all, the majority of existing approaches for indexing and storing RDF data attempts to find methods of reducing the space required to store the triples. For this aim we can exploit an idea that terms often repeat, or at least their substrings may often repeat across triples in a data set. Secondly, we focus on characteristics of triple components and their categorization. Suppose that we have a set of triples. Given a particular term (regardless its type), we may be interested in how many triples contain this term at a particular component (subject, predicate or object). Finally, we analyze so-called *star patterns* i.e., for each node we create a signature from a set of all ingoing and outgoing predicates and compare the size of star patterns

[9]http://linkeddata.org/

[10]http://acm.rkbexplorer.com/models/acm0proceedings.rdf

[11]http://downloads.dbpedia.org/3.7/en/persondata_en.nt.bz2

[12]An ontology describing persons, their relations, and activities (http://www.foaf-project.org/)

and frequency of their occurrence in analyzed data. Similarly, given a particular path, we can define its signature as a sequence of predicates connecting distinct nodes in the RDF graph.

The results show that although the data sets are from different areas, published by different methods and institutions, some of their characteristics are similar and, thus, the knowledge of these characteristics can be harnessed to make the management of RDF data more efficient. The complete metrics and especially all interesting results can be found in Section 3.

Contributions With *Analyzer* we fulfilled requirement R1, R3, and R4, i.e., the document acquisition, static analysis, and dynamic analysis. As the first use case we implemented and tested modules for analyses of real-world XML documents, XML schemas, XQuery queries, and RDF triples. In the first three steps of the analytic process we focused in more detail especially on issues of efficient crawling of XML data, re-validation of invalid XML documents, exploitation of similarity of XML data in data analysis, and XML query analysis. The main contributions are as follows:

- We introduce the architecture and functionality of *Analyzer*. To our knowledge it is a unique application that enables one to perform automatic, repeatable and extensible analyses of real-world data. Its basic version supports modules for XML data processing and analyses, however, using plugins it can be extended to any kind of data.
- *Analyzer* is a complex tool that supports not only the analytical part, but also various "supportive" functions, such as data crawling or data correction, as well as user-related features, such as definition of projects, visualization of results etc.
- With regard to the current support of XML, we study and describe four related issues – XML data crawling, XML data correction, structural analysis of XML data and query analysis of XML queries. In the former three cases we provide significant extensions to the current approaches, in the latter case we provide a unique approach which has not been considered in the current papers so far.
- We provide an overview of the current related work, in particular results of statistical analyses of real-world XML data. It enables us to show that all the results can easily be covered by *Analyzer*, further extended and repeated so that data changes and application evolution can be studied. Again, to our knowledge, such a feature has not been considered in recent literature so far.
- As a proof of the concept in a non-XML domain, we focused on several characteristics of publicly available Linked Data data sets. The results show that although the data sets are from different areas, published by different methods and institutions, some of their characteristics are similar and, thus, the knowledge of these characteristics can be harnessed to make the management of RDF data more efficient.

1.2.2 Data Extraction

As described in the previous sections, *Analyzer* focuses on documents which are imported by a user or automatically downloaded from the Web. The download module detects links in HTML or XML documents and it is not able to fulfill the requirement R2, i.e., extraction of data (e.g., URL of relevant pages, or related data stored in another document). Although it would be possible to implement an improved module with the ability to extract additional data from the Web and store them in a database, it does not correspond to the task *Analyzer* was originally created for. Thus we propose the following requirements for a new extraction tool:

1. documents are gathered from the Internet,
2. the full control over the extracted data is possible, i.e., the tool should not crawl the Web and extract data from different servers, but it should extract a high quality data from strictly specified servers,
3. the documents can be connected to other documents or entities, and
4. we want to share these extracted data (i.e., they should be available to others to create their own analyses).

For these purposes many different tools can be utilized and they use several ways to identify and extract the data from documents in various formats. We can categorize them by the criteria presented in [40]:

1. the level of automation,
2. the object of extraction, and
3. the domain specificity.

For example, the authors of [41] focus on localization of data-rich regions and they extract the relevant attribute-value pairs of records from Web pages across different sites, i.e., the method is automated (crawls different sources), the object of extraction is an HTML table and the results are not connected to a specific domain. On the other hand the *Ontology-Assisted Data Extraction* (ODE) [42] is a system for automatic identifying of lists of data on a Web page. The resulting fragments are then compared with an ontology and the data and labels are assigned (based on the maximum correlation).

Even though, there are many works focused on various aspects of data extraction, none of them covers all requirements. So we extended the original download module of *Analyzer* with focus on requirement R2, i.e., data extraction from structured or semi-structured documents. Additionally, we decided to support a concept which was designed to work with linked entities, i.e., Linked Data. This approach refers to a set of best practices for exposing, sharing, and connecting structured data on the Web. It is based on four main principles introduced by Time Berners-Lee [43]:

1. Use URIs[13] as names for things (resources).
2. Use HTTP[14] URIs in order that data consumers can look up those names.

[13]A uniform resource identifier (URI) is a string of characters used to identify a name or a Web resource.

[14]The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems (`http://tools.ietf.org/html/rfc2616`)

3. When someone looks up an URI, provide useful information, using RDF and SPARQL standards.

4. Include links to other URIs so that data consumers can discover more things (resources).

Consequently, we designed and implemented *Strigil* (see Section 5), a template based tool for data extraction. The templates are written in an XML script language inspired by XSLT (we decided to use our own language to provide more human readability and more extraction flexibility). The script is responsible for downloading of input documents, extraction of relevant data, post-processing (e.g., converting to suitable format, like date or decimal number), and mapping to ontological classes and attributes. The result of a processed script is an RDF graph that contains data extracted from all downloaded documents. Similarly as *Analyzer*, *Strigil* works with one initial URI of a document. In almost every script, it is possible to find more than one template. For example, we can consider a script, that will extract details about top 250 movies from the *IMDb*[15]. In this script, we need a template that will parse HTML document with list of those movies (and URL addresses of documents with details about movies), and a template, which will contain rules for data extraction from the document with details about the movie. The processing of these templates is depicted in Figure 1.4.

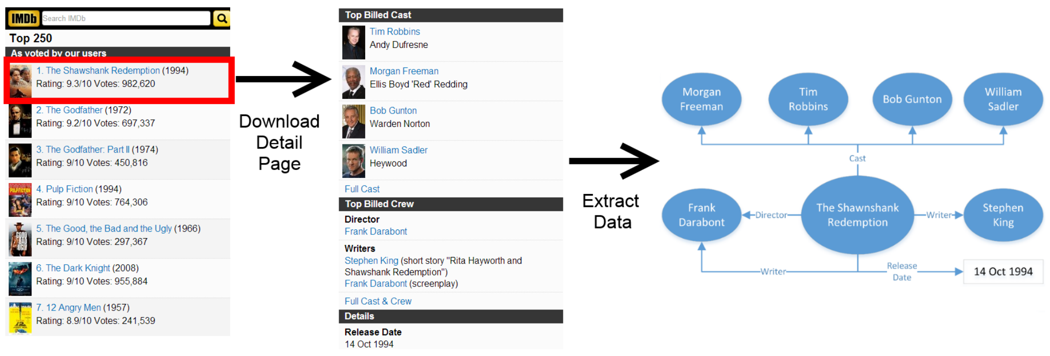


Figure 1.4: *Strigil* – template processing example

The crawling engine uses common HTTP features like cookies or forms to be able to get the data from the *Deep Web*, i.e., the data not available through direct links.

Note that *Strigil* is a part of a bigger framework (depicted in Figure 1.5) consisting of:

1. *Strigil* – a data extraction module extracting data based on an ontology

2. *ODCleanStore* – a data processing module cleaning, linking and enriching the data and a query execution module providing simple query end point for the data

3. *Payola* – a data visualization module allowing different views off the data

An important part of every template driven scraper is the selected template language. As mentioned before, in *Strigil* we decided to propose a script language, inspired by XSLT, to transform different documents to data specified by ontologies. The script language uses selectors to specify data fields in a source document which are

[15]Internet Movie Database (`http://www.imdb.com`)

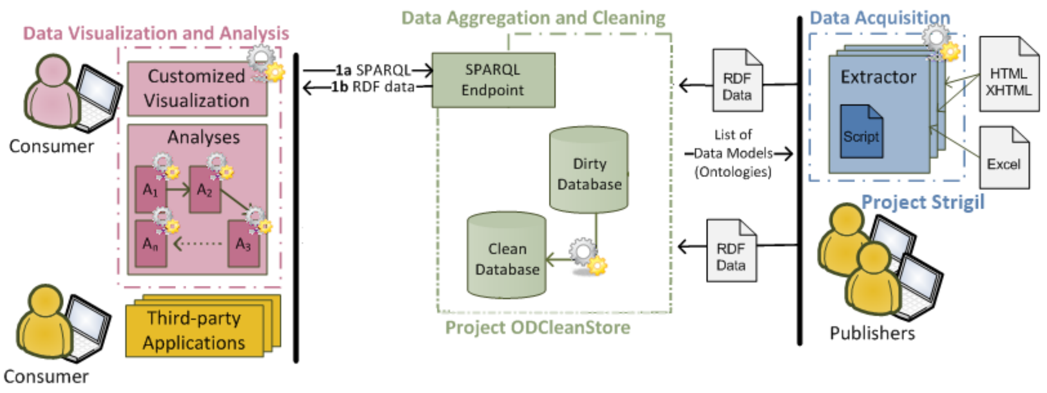
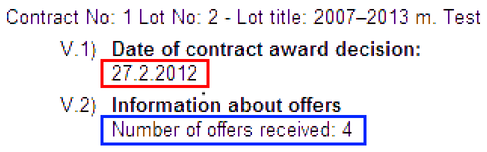


Figure 1.5: Architecture of framework formed by *Strigil*, *ODCleanStore*, and *Payola* tools

```
<scr:onto-elem>
    <scr:value-of
      select="div\#date"
      property="http://purl.org/procurement/public-
         contracts#tenderDeadline"  />
</scr:onto-elem>
```

Listing 1.1: *Strigil* scraping language – an example of element `onto-elem`

directly connected to ontology classes or properties. Currently we work with *JSoup* selectors[16] for HTML documents and our custom selectors for Excel files (similar to HTML selectors, extended with some constructs to work with work sheets). For example, we have an HTML page fragment depicted in Figure 1.6 with two outlined elements `div` with attributes `id` with values *date* (outlined in red) and *offers* (outlined in blue) respectively.



Contract No: 1 Lot No: 2 - Lot title: 2007–2013 m. Test
V.1) Date of contract award decision:
27.2.2012
V.2) Information about offers
Number of offers received: 4

Figure 1.6: HTML page fragment

A *JSoup* selector for the red element `div` could be

`div#date.`

This selector can be used in the *Strigil* script language to extract content of the element and assign its value to a RDF blank node as depicted in Listing 1.1. In particular, the element `scr:onto-elem` declares an RDF node with one property represented by element `scr:value-of`. The attribute `property` declares the URI of the property and the attribute `select` specifies the used selector.

[16]`http://jsoup.org/cookbook/extracting-data/selector-syntax`

Since, the HTML documents on the Web can change, *Strigil* offers additional supportive functions like repetitive downloads in user-specified intervals or automatic checking of validity of used selectors. In other words, *Strigil* compares the number of used selectors and the number of returned empty responses and it notifies the user if the ratio is bellow a user-defined threshold. In addition, although we have described selectors for HTML documents, the selectors can be used to extract data from other formats depending on available implementations.

The architecture and the scraping language is described in detail in Section 5.

Contributions In this part of the thesis we addressed mainly the requirement R2, i.e., extraction of data from different documents. The main contributions we proposed are as follows:

- We propose the architecture of *Strigil*. It is a tool for data extraction from semi-structured documents, e.g., HTML or spreadsheets. Although we mentioned only these formats, the system is designed to be easily extended to work with any type of documents with proper implementation of selectors.
- Contrary to other approaches which are universal but not user-friendly, we focus on simplicity of the user interface which allows a less technically experienced user to create a scraping script. On the other hand *Strigil* was designed with regard to simple extensibility of input formats.
- Additionally, *Strigil* contains a support for mapping to ontologies describing a domain or a combination of different domains which allows the user to work with specified data types without their explicit declaration in the script.
- *Strigil* is a template driven system (contrary to adaptive approaches which identify the object of extraction heuristically), i.e., it does not try to adapt extraction script to new or changed documents. But, since the documents on the Web often change, we implemented a simple check to allow a user to repair the scraping script.

1.2.3 Schema Inference

As we have already mentioned, various statistical analyses of real-world XML data show that a significant portion of XML documents (in particular 52% [6] of randomly crawled or 7.4% [14] of semi-automatically collected) still have no schema at all. What is more, XML Schema definitions (XSDs) are used even less (only for 0.09% [6] of randomly crawled or 38% [14] of semi-automatically collected XML documents). Consequently a new research area of automatic construction of an XML schema has opened. The key aim is to create an XML schema for the given sample set of XML documents that is neither too general, nor too restrictive. It means that the set of document instances of the inferred schema is not too broad in comparison with the sample data but, also, it is not equivalent to it. Currently, there are several proposals of respective algorithms, but there is also still a space for further improvements. In particular, since according to Gold's theorem [11] regular languages (i.e., general XML schemas) are not identifiable only from positive examples (i.e., sample XML documents which should be valid against to the resulting schema), the existing methods need to exploit either heuristics or a restriction to an *identifiable* subclass of regular languages.

On the basis of these observations, in the last part of this thesis we focus on a schema inference. Currently there are several well-known algorithms, but there is still a space for further improvements. To enable combining of existing approaches and their extensions with new features, we proposed and implemented *jInfer*, a general tool for XML schema inference. It is an easily extensible tool that enables implementation, testing and comparison of new modules of the inference process. Note that a similar system called *SchemaScope*, was described in [44]; however, its main target are grammar-inferring approaches, especially those proposed by its author. In *jInfer* we focus on a more general view of the problem, involving mainly the heuristic approaches.

The inference process in *jInfer* consists of three consecutive steps:

1. *Initial grammar* (IG) generation converting inputs (i.e., XML documents, and old obsoleted XML schema, etc.) to IG representation. All documents, schemas and queries selected as input are evaluated, simple rules are extracted and sent to the next step. For example, the translation of an XML document fragment to IG rules is depicted in Figure 1.7.

```
<person>
  <info>
    Some text
    <note/>
  </info>
  <more/><more/><more/>
</person>
<more/>
<person>
  <more/>
</person>
```

$$\longrightarrow$$

$$\begin{array}{rcl}
person & \rightarrow & info, more, more, more \\
info & \rightarrow & simple\_data, note \\
note & \rightarrow & empty\_concatenation \\
more & \rightarrow & empty\_concatenation \\
person & \rightarrow & more, more, more \\
more & \rightarrow & empty\_concatenation \\
more & \rightarrow & empty\_concatenation \\
more & \rightarrow & empty\_concatenation
\end{array}$$

Figure 1.7: XML fragment converted to initial grammar

   Note that in this case the process is very simple. However, complications can bring new input data, such an obsolete schema, queries etc.

2. *Simplification* of IG comprising clustering of production rules, building the output grammar, inference of simple data types, or inference of integrity constraints. For example, the previous rules (clustered according to element name) could be simplified to a single rule for element `person`:

$$person \rightarrow info?, more\{1,3\}$$

   Rules for elements `info`, `more` and `note` after simplification will be:

$$\begin{array}{rcl}
info & \rightarrow & simple\_data, note \\
note & \rightarrow & \lambda \\
more & \rightarrow & \lambda
\end{array}$$

3. *Export* to a selected schema language (e.g., DTD, XML Schema, etc.). The result of this step is a textual representation of the schema, which is sent back to the framework (and later displayed, saved etc). For the previous simplified rules, the resulting DTD would be:

```
<!ELEMENT person
          (info?, more, more?, more?)>
<!ELEMENT info (#PCDATA | note)*>
<!ELEMENT note EMPTY>
<!ELEMENT more EMPTY>
```

Note that the process is not so straightforward. For instance, for element `person`, when the simplified grammar specifies its occurrence to at least once and at most 3 times, as DTD has no such construct, the export module has to find a suitable expression.

The overall *jInfer* architecture is depicted in Figure 1.8.

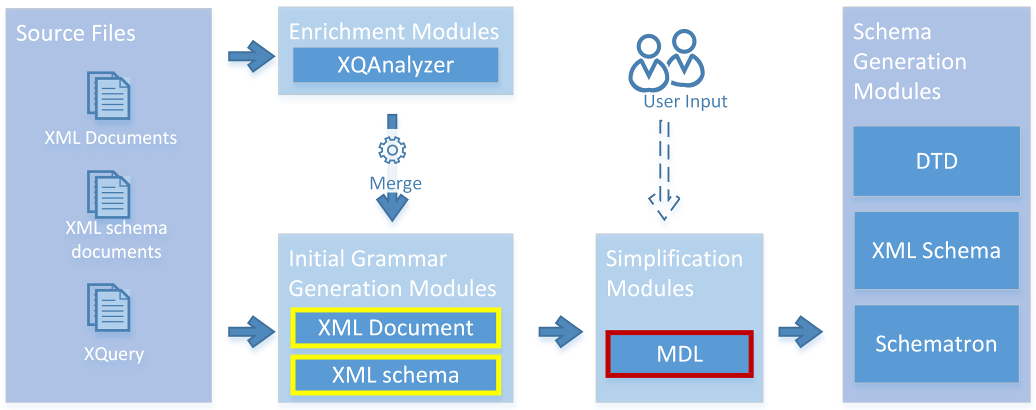


Figure 1.8: *jInfer* – initial grammar creation and simplification modules

Like in the previous systems, also in *jInfer* these steps are implemented as standalone modules, so it is possible to easily replace and test different approaches. In this thesis we describe selected improvements of each of these steps.

Initial Grammar Creation Firstly we propose an approach to use an old obsolete schema combined with XML documents to create an IG (related modules are outlined in yellow in Figure 1.8). We follow the steps proposed in [45], i.e., positive examples (element instances from XML documents) are first clustered by element names, context and content. Contrary to existing works, we parse XML schema files into grammar rules and cluster them by element name together with element instances originating from XML documents. Since in both XML Schema and DTD the element content model is basically specified by a regular expression, we consider positive examples as being generated by a *deterministic probabilistic finite automaton* (DPFA) and try to infer this automaton. Then it is modified by merging its states. To select the states to merge we use two verified state equivalence criteria: *sk-strings* [46] criterion and *k,h-context* [47] criterion. In particular, we implemented four general merging strategies (the related module is outlined in red in Figure 1.8):

1. *Greedy* simply merges all candidate states provided by criterion testers.

2. *GreedyMDL* uses the MDL[17] principle [48] to evaluate a DPFA and input strings encoded by the automaton. While trying to merge candidate alternatives, the

[17]The minimum description length (MDL) principle is a formalization of Occam's Razor in which the best hypothesis for a given set of data is the one that leads to the best compression of the data.

GreedyMDL strategy always keeps the currently achieved minimum quality value (and the associated automaton). A space of possible solutions is explored in a greedy way, but a sort of a complete scanning of continuation possibilities is done: all candidate alternatives to merge are evaluated. The algorithm stops when there are no more candidates to merge, or when all alternative candidates returned by merge criterion testers end up in an automaton with higher quality value than the one actually achieved.

3. *HeuristicMDL* works in similar way as GreedyMDL but it holds *n* best minimal solutions instead of only one.

4. *DefectiveMDL* is our original proposed strategy that tries to decide which input strings are so eccentric that they probably are mistakes and should be repaired in input documents rather than incorporated into output schema.

The inferred automaton is then converted into an equivalent regular expression using a *state removal algorithm* [49] and the regular expression is added to the *output grammar*. Our proposed MDL metric and implemented strategies are described in Section 6.6.

Data Types and Integrity Constraints Secondly we focus on improvement of the inference process with additional input information, namely XQuery queries to infer XML data types and keys. The implemented module is outlined in Figure 1.9. The proposed extension consists of four main steps:

1. Construction of syntax tree of each XML query – We use lexical and syntax analyses introduced as an *Analyzer* module and proposed in Section 4 and for each XQuery on input, we construct a data structure called a syntax tree.

2. Static analysis of expression types – The algorithm searches for particular expressions in the syntax trees and statically (without query evaluation) determines their types.

3. Inference of built-in types – When the types of expressions are determined, the selected forms of expression are utilized to infer types of elements and attributes.

4. Key discovery – The final step is an extension of approach [50] inferring keys and foreign keys.

As we have mentioned, for the purpose of query analysis, we use syntax tree construction proposed in Section 4. Then the algorithm searches for particular expressions in the syntax tree to determine their types which are utilized to infer types and occurrence modifiers of elements and attributes. For example, if the operator in an expression is one of $+$, $-$, div, mod, $*$, $/$, one operand is a PathType *P* = `//person/more` and the other operand is one of numeric built-in types *T* = *Integer*, i.e.,

$$\texttt{//person/more} + 5$$

then the inferred production rule is

$$\texttt{//person/more} \rightarrow Integer.$$

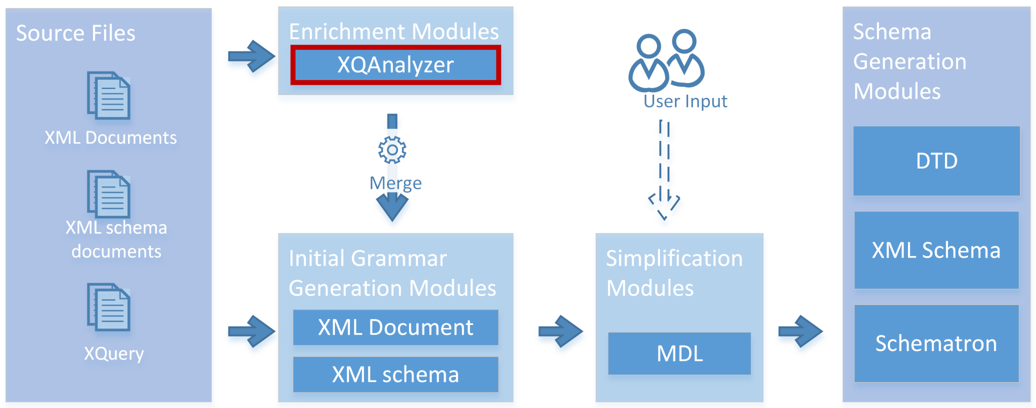


Figure 1.9: *jInfer* – data types and integrity constraints module

Finally, we extend the approach from paper [50]. They suppose that each join is done via a key/foreign key pair. Thus the query is searched for so-called *join patterns*. The algorithm recursively, in pre-order, searches the syntax tree of the XML query, every node representing a FLWOR expression is processed and classified using application of six proposed rules according to used aggregation functions. Based on these rules the algorithm decides which is the key and which is the foreign key. The details about the patterns and rules are described in Section 6.7.

Schematron Export The last but not least proposed extension aims at inferring of a completely new type of output. We describe an approach for inference of a Schematron schema. *Schematron* [51] uses a completely different strategy for XML schema definition than the other so-called grammar-based approaches. However, on the other hand, it probably served as an inspiration for XML Schema assertions. As we can see in Figure 1.10, each Schematron schema is based on the idea of *patterns*. A pattern can be described as a set of *rules* an XML document must satisfy to be valid against a Schematron schema. A rule involves either element `assert` or `report` depending on the requirement of satisfaction or not satisfaction of the condition specified in attribute `test`.

In the described solution we specify constraints using XPath 1.0 since it is widely spread. The algorithm was implemented as a module of *jInfer* (outlined in Figure 1.11). We divide the transformation process of a production rule into three steps:

1. *Generate the correct context* for each rule.

2. Check the correct *sum of children* utilizing `child` axis and `count` function.

3. Match the *order of children* with the XPath axes `preceding` and `following`.

The correct context for rules is necessary for the algorithm to work correctly and it is used for constraints generated by steps 2 and 3. In particular, we use one of the following methods:

- *Trivial solution* – In this case, we can create a simple relative XPath expression for each production rule h of the form $A \to aR$, where A is a non-terminal, a is a terminal and R is a regular expression, using only the name of terminal a. For example, the relative XPath context for non-terminal *Person* is `//person`

```
<schema xmlns="http://www.ascc.net/xml/schematron">
 <pattern name="Conditions for list of employees">
  <rule context="employees">
   <assert test="person">No person found</assert>
  </rule>
 </pattern>
 <pattern name="Conditions for one person">
  <rule context="person">
   <assert test="name">A person must have an
    element name</assert>
   <report test="name[2]">A person cannot have
    multiple elements name</report>
   <assert test="@id">A person must have an
    attribute id</assert>
   ...
  </rule>
 </pattern>
 ...
</schema>
```

Figure 1.10: An example of a Schematron schema

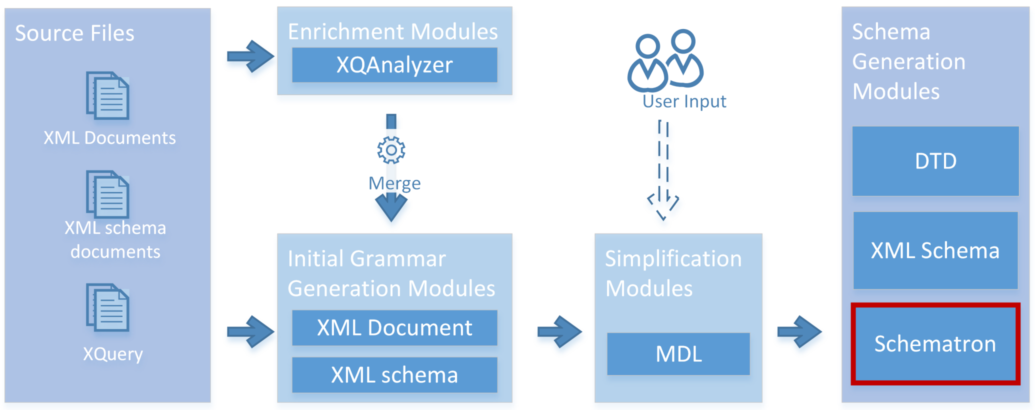


Figure 1.11: *jInfer* – Schematron export module

- *K-ancestors* – This method is used for schema inference in [12]. The key idea is to identify the context based on the element name and the name of K closest ancestors, whereas according to the authors more than 98% of context matching could be expressed with this solution and K equal to 2 or 3. For example, for $K = 2$ the XPath context for non-terminal *Person* is `//database/person`

- *Absolute Path without Recursion* – A more reliable, less restricted way to identify the correct context can be specified using the absolute paths. For example, the XPath context for element *Data* is `/database/person/data`

- *Simple Recursion in Production Rules* – This algorithm creates an XPath expression that matches continuous sequence (which may not be limited) of elements in the parent/child relation. Since we have only simple recursion and thus the repeating sequence consists of only a single terminal, our work is relatively simple. We can use the `descendant-or-self` axis with a constraint on the ancestors.

- *Recursion with Deterministic Content* – The algorithm creates a complex expression that matches the correct leading element and checks that there are no other elements, that no element is missing and that the elements are in correct order.

In the second step we focus on validation using minimum and maximum occurrence checks – *boundary rules* – of elements from a production rule. We process all production rules and count the minimal and maximal occurrence of the elements. We then use the XPath function `count` to check that there are no illegal children.

The third step ensures the order of the children elements. Since XPath 1.0 does not support regular expressions, we can construct several rules that use XPath axes (`preceding`, `following`, etc.) and express the regular expression with them. The basic idea results from [52] and it creates constraints for allowed following siblings. A more complex regular expression may require more rules to express it.

The complete description of Schematron schema generation is described in Section 6.8.

Contributions In this part of the thesis we addressed the requirement R5, i.e., data processing. The key contributions can be summed up as follows:

- We propose the architecture of *jInfer*. It represents an easily extensible tool that enables one to implement, test and compare new modules of the inference process.
- Since the compulsory parts of the process, such as parsing of XML data, visualization of automata, transformation of automata to XML schemas etc. are implemented, the user can focus purely on the research area and the improved aspect of the inference process.
- We describe and implement a module to use an old schema to generate initial grammar and described usage of MDL metric in different merging strategies.
- We optimize the inference process with a new type of input (in this case XQuery queries) and with regard to classical inference approaches completely new output (in this case simple data types and simple integrity constraints).
- We design and implemented a new module for expressing output grammar in Schematron language, being probably the first approach of this kind ever proposed.

1.3 Related Papers

The list of author's publications as follows:

Journal Articles

[53] J. Stárka, M. Svoboda, J. Sochna, J. Schejbal, I. Mlýnková, and D. Bednárek, "Analyzer: A Complex System for Data Analysis," *The Computer Journal*, vol. 55, pp. 590–615, May 2012. IF: 0.785, 5-Year IF: 0.943

Full Conference Papers

[54] J. Stárka, M. Svoboda, and I. Mlýnková, "Analyses of RDF Triples in Sample Datasets," in *COLD '12: Proceedings of the 3rd International Workshop on Consuming Linked Data of ISWC '12: 11th International Semantic Web Conference* (J. Sequeda, A. Harth, and O. Hartig, eds.), vol. 905 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2012

[55] M. Kozák, J. Stárka, and I. Mlýnková, "Schematron Schema Inference," in *Proceedings of the 16th International Database Engineering & Applications Sysmposium*, IDEAS '12, (New York, NY, USA), pp. 42–50, ACM, 2012

[56] M. Klempa, J. Stárka, and I. Mlýnková, "Optimization and Refinement of XML Schema Inference Approaches," in *Proceedings of the 3rd International Conference on Ambient Systems, Networks and Technologies (ANT)*, vol. 10, (Niagara Falls, Canada), pp. 120–127, Elsevier, 2012

[57] M. Mikula, J. Stárka, and I. Mlýnková, "Inference of an XML Schema with the Knowledge of XML Operations," in *Proceedings of the 8th International Conference on Signal Image Technology and Internet Based Systems (SITIS)*, pp. 433–440, IEEE Computer Society Press, 2012

[58] J. Schejbal, J. Stárka, and I. Mlýnková, "XQConverter: A System for XML Query Analysis," in *Proceedings of the 6th International Workshop on Flexible Database and Information System Technology of DEXA '11: 22nd International Conference on Database and Expert Systems Applications*, (Toulouse, France), pp. 129–133, IEEE Computer Society Press, 2011

[59] J. Stárka, I. Mlýnková, J. Klímek, and M. Nečaský, "Integration of Web Service Interfaces via Decision Trees," in *Innovations in Information Technology (IIT), 2011 International Conference on*, pp. 47 –52, IEEE, april 2011

[60] M. Svoboda, J. Stárka, J. Sochna, J. Schejbal, and I. Mlýnková, "*Analyzer*: A Framework for File Analysis," in *Proceedings of the 2nd International Workshop on Benchmarking of Database Management Systems and Data-Oriented Web Technologies of the 15th International Conference on Database Systems for Advanced Applications*, vol. 6193 of *LNCS*, (Tsukuba, Japan, April 1-4), pp. 227–238, Springer Berlin / Heidelberg, 2010

Local Conference Papers

[61] M. Svoboda, J. Stárka, and I. Mlýnková, "On Distributed Querying of Linked Data," in *Proceedings of the 11th Workshop DATESO 2012*, pp. 143–150, MATFYZPRESS, 2012

[62] J. Stárka, M. Svoboda, J. Schejbal, I. Mlýnková, and D. Bednárek, "XML Document Correction and XQuery Analysis with Analyzer," in *Proceedings of the 10th Workshop DATESO 2011*, (Ostrava – Poruba, Czech Republic), pp. 61–72, VSB – Technical University of Ostrava, 2011

Since the area of document acquisition and processing is wide, we continue in our research. We have submitted two journal papers which describe related topics in more detail. Additionally, we have one conference paper under review process presenting our last related research.

Submitted Journal Articles

[63] M. Klempa, M. Kozák, M. Mikula, R. Smetana, J. Stárka, M. Švirec, M. Vitásek, M. Nečaský, and I. Holubová, "jInfer: a Framework for XML Schema Inference," *The Computer Journal*. *(Note: paper under review process)*

[64] M. Nečaský, J. Klímek, T. Knap, J. Mynarz, J. Stárka, and V. Svátek, "Linked Data Support for Filing Public Contracts," *Computers in Industry, Special issue on New trends on E-Procurement applying Semantic Technologies*, 2013. *(Note: paper under review process)*

Submitted Conference Papers

[65] J. Stárka and I. Holubová, "Strigil: A Framework for Data Extraction," *ODBASE 2013*, 2013. *(Note: paper under review process)*

1.4 Road Map

The rest of the thesis is structured into chapters according to individual contributions as published in our research papers. The order of the papers is based on the problems described earlier.

- In Section 2 we propose *Analyzer*, the framework for document analysis. We describe its architecture, main modules and we present the results computed over real-world XML data sets.
- Section 5 describes *Strigil*, a system for data extraction. We show the main advantages of the system and propose a scraping language used to select data from different types of documents.
- In Section 3 we provide characteristics that appropriately capture and describe features of RDF triples and experimental results over a few selected real-world RDF data sets.
- In Section 4 we focus on static analysis of XQuery programs and their usage in the inference process.
- In Section 6 we describe *jInfer*, the framework for automatic schema inference with different approaches across all aspects of the inference process.

Table 1.1 summarizes the research topics. Every research topic is supplemented with the related requirement, the section of the thesis where the topic is targeted, and relevant author's publications introduced in Section 1.3.

Note that not all author's publications mentioned in Section 1.3 are relevant to this thesis. In paper [59], we utilize similarity metrics to find an automatic mapping of XML formats to a conceptual XSEM model [67]. In paper [61], we describe a querying system for Linked Data.

Relevant topic	Requirement	Section	Author’s Publications
Document Acquisition and Analysis	R1, R3, R4	2	[60], [53]
Data Extraction	R2	5	[65], [64]
Static Analysis	R3		
– XML documents and schemas		2.6	[53], [62]
– XQuery		2.7.3, 4	[58]
– Linked Data		3	[54]
Data Inference	R4	6	[56], [55], [66], [57]

Table 1.1: Topics summary

2. Document Extraction and Analysis

In this section we introduce Analyzer *– a complex framework for performing statistical analyses of real-world documents. Exploitation of results of these analyses is a classical way how data processing can be optimized in many areas. Although this intent is legitimate, ad hoc and dedicated, analyses soon become obsolete, they are usually built on insufficiently extensive collections and are difficult to repeat.* Analyzer *represents an easily extensible framework, which helps the user with gathering documents, managing analyses and browsing computed reports.*

The contents of this section was published as a journal paper Analyzer: A Complex System for Data Analysis *[53] in the Computer Journal (IF: 0.785, 5-Year IF: 0.943) and it extends our previous conference papers* Analyzer: A Framework for File Analysis *[60] and* XML Document Correction and XQuery Analysis with Analyzer *[62].*

2.1 Introduction

The eXtensible Markup Language (XML) [1] 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. Firstly, XML was only exploited as a syntax for parametrization files. However, with the growing popularity of advanced XML technologies (such as XML Schema [23, 68], XPath [28], XQuery [21], XSLT [69] etc.) there appeared also true XML applications that exploit the whole family of XML standards for managing, processing, exchanging, querying, updating, and compressing XML data that mutually compete in speed, efficiency, and minimum space and/or memory requirements. Similarly, there occurred a huge amount of standards based on XML technologies, such as WSDL [70], SVG [71], RDF [72], OpenOffice [73] etc., that exploit the advantages of XML for specific purposes. Unfortunately, for majority of these techniques and applications there can be found a number of drawbacks concerning their efficiency.

Under a closer investigation we can distinguish two situations. On the one hand, there is a group of general techniques that take into account all possible features of input XML data. This idea is obviously correct, but 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. From the point of view of structural or space efficiency, it can even be harmful. On the other hand, there are techniques that somehow do restrict features of the given input XML data. For them it is natural to expect inefficiencies to occur only when the given data do not correspond to these restrictions. (In extreme cases, selected approaches even do not support any other data than those that they can process efficiently.) The problem is that such restrictions do not result from features of real-world XML applications and requirements, but they are often caused by limitations of a particular technique, complexity of such a solution, irregularities etc. Consequently, the restrictions are not natural and do not reflect user requirements.

We can naturally pose two apparent questions:

1. Is it necessary to take into account a feature that will be used minimally or will not be used at all?
2. If so, what are these features?

The answer for the first question obviously depends on the particular situation, i.e., application. The second question can be answered only using a detailed analysis of a sample set of real-world XML documents. However, working with real-world data is not simple, since they can often change, are not precise, or even involve a number of errors. In our approach, we have addressed the following four problems:

- *Data crawling* – there exists a huge number of Web crawlers; however, their filters and crawling strategies must be retargeted from HTML to the XML family of documents.
- *Processing of incorrect data* – since the data are usually human-written, they contain 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*.
- *Structural analysis* – a number of features like size, depth, or fan-out may be statistically examined in a collection of XML documents; similar analysis may also be applied to XML schemas in any form.
- *Query analysis* – a collection of XQuery, XSLT, or XPath queries may be examined for the presence of certain query language constructs or their combinations. Because of the complexity of the query languages and the large variety of motivations for such examination, the analytical tool must be as generic as possible.

In addition, we have to cope with the fact that the data can change and, hence, the analytical phase must be repeatable and extensible. And, finally, having obtained the results of the statistics, we need to be able to visualize and analyze the huge amount of information efficiently and mutually compare the results.

In this section we describe a proposal background, architecture outline, implementation aspects and usage scenarios of a general framework called *Analyzer* that aims to cope with all the previously named requirements. In other words, it provides all essential functionality for an easy management of files to be analyzed, configuration and execution of selected analyses and an advanced graphical user interface (GUI) for browsing generated reports. The key advantage of *Analyzer* is extensibility. This not only means the ability to implement own and more suitable kernel components responsible, e.g., for storing computed analytical data, but primarily the open concept of plugins. *Analyzer* provides a general environment, whereas all analytical computations themselves are defined solely within the implementation of plugins. The user is therefore expected to first install *Analyzer* itself and then create his/her own plugins designed to correspond to the determined research intents. Although our initial motivations were related to XML data, *Analyzer* usage is not limited only to this area.

Contributions The key contributions of this section can be summed up as follows:

- We introduce the architecture and functionality of *Analyzer*. To our knowledge it is a unique application that enables one to perform automatic, repeatable and extensible analyses of real-world data. It currently supports modules for XML data processing and analyses, however, using plugins it can be extended to any kind of data.

- *Analyzer* is a complex tool that supports not only the analytical part, but also various "supportive" functions, such as data crawling or data correction, as well as user-related features, such as definition of projects, visualization of results etc.

- With regard to the current support of XML, we study and describe four related issues – XML data crawling, XML data correction, structural analysis of XML data and query analysis of XML queries. In the former three cases we provide significant extensions to the current approaches, in the latter case we provide a unique approach which has not been considered in the current papers so far.

- We provide an overview of the current related work, in particular results of statistical analyses of real-world XML data. It enables us to show that all the results can easily be covered by *Analyzer*, further extended and repeated so that data changes and application evolution can be studied. Again, to our knowledge, such a feature has not been considered in recent literature so far.

Relation to Previous Work In this paper we partially exploit, combine and, in particular, extend our several previous results. Motivated by a successful and interesting statistical analysis of real-world XML data [14], *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 [74]. Its first release 1.0 involved only basic functionality to demonstrate its key features and advantages and it was briefly introduced in paper [60]. Its four key parts were then extended in four master theses of its authors supervised by Irena Mlýnková and David Bednárek. In particular, Jan Sochna [75] focussed on the primary aspect of *Analyzer* – efficient crawling of XML data. Martin Svoboda [76] proposed several improvements of algorithms for correction of XML data. Jakub Stárka [77] focussed on an important aspect of the structural analysis of XML data, i.e., XML similarity. And, finally, Jiří Schejbal [78] dealt in his thesis in current most open topic of analysis of XML operations, i.e., XQuery queries. In the following text we will describe and put into context the key results of the four theses and, in particular, show their close connection and resulting advantages and contributions they bring. Our aim is to provide a throughout description of all aspects of *Analyzer* as well as data analysis in general, useful for both future users of *Analyzer* and researchers dealing with related issues.

2.2 Motivation

The idea of exploitation of the knowledge of real-world data is not new and currently we can find several applications and use cases, where it is successfully applied. From the general point of view we are interested in the subset of constructs and structures allowed by a particular language that is commonly used in practice. In this section we provide several examples, where the knowledge of real-world data has been successfully exploited.

Incorrect Assumptions on Real-World Data 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 two distinct cases – schema-driven XML-to-relational storage strategies and exploitation of recursion.

Schema-driven XML-to-relational mapping methods [79, 80] are based on an existing schema S_1 of stored XML documents which is mapped to an (object-)relational database schema S_2. The data from XML documents valid against S_1 are then stored into relations of S_2. The purpose of these methods is to create an optimal schema S_2, which consists of reasonable amount of relations and whose structure corresponds to the structure of S_1 as much as possible. Naturally, such approaches require a presence of an XML schema. But, statistical analyses of real-world XML data show that a significant portion of XML documents (52% [6] of randomly crawled or 7.4% [14] of semi-automatically collected) still have no schema at all. What is more, XML Schema definitions (XSDs) are used even less (only for 0.09% [6] of randomly crawled or 38% [14] of semi-automatically collected XML documents) and even if they are used, they often (in 85% of cases [5]) define so-called *local tree grammars* [81], i.e., grammars that can be defined using DTD as well. Hence, these methods need to be accompanied with approaches for inference of an XML schema for a given set of XML documents [45, 82].

Conversely, the support for recursion is often neglected and it is considered as a side/auxiliary construct. However, analyses show that in selected types of XML data it is used quite often (in 58% of all DTDs [2], or in 43% of document-centric and 64% of exchange documents [14]) and, hence, its 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.

Efficient Processing of XML Data with Limited Complexity Another important observation common to all the current papers describing results of statistical analyses of real-world data is that the data is usually much simpler than the respective standard allows. Such feature opens a wide range of optimization strategies that do not have to consider the full generality of the standard, but can count on some limitations. One of the most surprising observations of this kind is that the average depth of XML documents is less than 10, mostly around 5. This information is already widely exploited in techniques [7, 83] which represent XML documents as a set of points in multidimensional space and store them in corresponding data structures, e.g., *R-trees*, *UB-trees* [84], or *BUB-trees* [85], for the purpose of efficient querying. The dimension of the space is given by the depth of the data, so the lower, the better.

Restriction to Real-World Data Structures A situation similar to the previous one occurs in cases when we can restrict a particular approach only to cases which occur in real-world data and, hence, have a reasonable basis. A research area which widely exploits the knowledge of complexity of real-world data is inference of XML schemas from a given sample set of XML documents. Since according to Gold's theorem [11] regular languages (i.e., those generated by XML schemas) are not identifiable only from positive examples (i.e., sample XML documents which should be valid against the resulting schema), the existing methods need to exploit either heuristics or a restriction to an *identifiable* subclass of regular languages. The question is which subclass should be considered. The authors of papers [12, 13] result from their analysis of real-world XML schemas [5] and define the classes so that they cover most of the real-world examples. So the identifiable subclass corresponds to a realistic situation.

Tuning of Weights and Parameters Last but not least typical example of exploitation of characteristics of real-world XML data can be found in reasonable and realistic setting of various weights, parameters and characteristics of different approaches. As an example we can consider two XML applications – evaluation of similarity of XML schemas and adaptive XML-to-relational storage strategies.

Similarity of XML schemas is currently exploited in many approaches, typically as a kind of optimization heuristics. The current approaches [8, 9, 10] are based on the idea of exploitation of various *matchers*, i.e., functions which evaluate similarity of selected simple data characteristics (e.g., similarity of element names, similarity of number of subelements etc.). Their results are then aggregated to a resulting *composite similarity measure* using a kind of weighted sum. The problem is how to set the weights so that the result reflects the reality. And the solution can be found in statistical analyses of real-world data.

In case of adaptive XML-to-relational storage strategies we need to solve a similar problem. The approaches [86, 87] focus on the idea that each application requires a different storage strategy to achieve optimal efficiency. So, before they provide the resulting mapping, they analyze a given set of sample data and operations which represent the target application and adapt the resulting mapping accordingly. Hence, again, an analysis of real-world data is crucial so that the algorithm works correctly and efficiently.

Apparently, in all the described examples we need to know the structure and complexity of the real-world data as precisely as possible. What is more, since user requirements often change and new applications occur every day, we also need to know whether and how the data characteristics evolve and adapt the approaches and optimizations respectively.

2.3 Framework Architecture

This section concerns with the architecture of *Analyzer*, proposed analytical model and basic implementation aspects.

2.3.1 Framework Architecture

The implementation of *Analyzer* allows the user 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 2.1. The first one contains components, which are shared by all the projects. The second one represents components exclusively used and created in each opened project separately.

Project Components Components at the project level involve particularly *repositories*, *storages* and *crawlers*. They are exclusively owned by each project, but this does

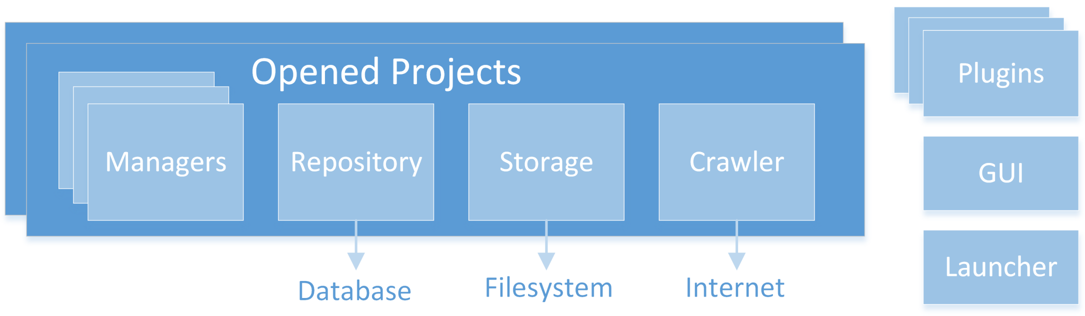


Figure 2.1: Architecture of *Analyzer*

not mean that, e.g., a real relational database server behind a repository cannot be used by multiple projects. This is allowed and a given component only has to ensure the required isolation between individual projects (which can be done easily in the given example using different databases).

First, each project must have a single *repository*. It serves for storing all computed analytical data and the majority of project configuration metadata. Although the design of *Analyzer* does not require it, all provided repository implementations are based on standard relational databases. Second, *storages* are used for storing document contents, i.e., binary contents of analyzed files. The only stable implementation is based on a native file system, but experiments were taken with native storages for XML documents, too. Third, there are two ways how we can insert files to be analyzed into a created project. First, we are able to import them from a specified storage, or we can use *crawlers* to download them from the Web. The download process may involve accessing of explicitly required files, or the crawler itself may be able to attempt to find other referenced files using link traversal, limited only by a maximum allowed searching depth.

An essential design feature of all these three components is extensibility. Although a typical user would probably not need it, new components can be implemented and added relatively easily into the entire system.

The project layer also contains a set of *managers*, which are responsible for creating, editing and processing of all entities such as *documents*, *collections* or *reports*. As all computed analytical data are stored permanently in a repository, in order to increase efficiency, the managers are able to cache loaded data and release them, if they are no longer required at runtime. Some managers are also able to postpone and aggregate update operations, but the consistency of computed data is still guaranteed.

Shared Components Shared components are instantiated only once in a running *Analyzer* application and are used by all opened projects together. Passing over auxiliary components, the most important one is a *launcher*, which is responsible for executing *tasks* over all such projects. *Tasks* represent small units of analytical or other actions and computations. For example, each download of a selected file or computation over a given document is internally encapsulated and processed in a form of a task. Once it is decided that some work should be done, a new task is created, scheduled and later on prepared for execution, if no blocking dependencies exist. The execution itself is invoked by the *launcher*, maintaining a parallel environment with worker threads prepared in a pool. If a given task could not be successfully finished (for whatever reasons), launcher attempts to execute it repeatedly with a defined number of attempts.

Clearly, this model is a compromise, since it decreases the efficiency, but enables nearly full control over project processing. The user is able to attach/detach a particular project to/from the launcher and, thus, say whether tasks from the project should be executed at the moment, or not. As a consequence, the user can pause started computations and resume them later.

Graphical User Interface The GUI is based on possibilities of the NetBeans platform [88]. It brings the complete and robust environment for creating and managing projects and performing analyses from their configuration to browsing of the computed reports .

The browser itself contains adjustable windows through which the user is able to monitor the progress of computations and browse all existing entities in opened projects. The data are provided mainly in a form of interactive *trees* or *listings*. More complicated project actions, like creation of new components or configuration of analyses, are implemented using *wizards*.

2.3.2 Analysis Model

In this section, we describe the life cycle of a typical analysis. As an example, suppose we want to develop an indexing service over Linked Data [89] and we want to know their basic characteristics, e.g., maximum and average depth or average number of child elements, to optimize our service. We choose DBpedia[1] as a source of data, create a project and set up all components described in the previous section. Documents are stored locally and the computed results in a MySQL database. For this analysis, we discard all documents except for RDF ones. Additionally, we add plugins to get expected results which have to be able to identify the RDF files, repair them and compute expected characteristics. When *Analyzer* downloads the files the analyzed measures are computed and stored in the repository. Then, we create a collection with all documents, *Analyzer* computes aggregated results and we can use the results for the optimization.

Later on, we decide to extend the service to index over another domain, e.g., `data.gov.uk`. We can use our old project and download and analyze new version of the documents from DBpedia and `data.gov.uk`. We add a new collection with newly added files from DBpedia to get the changes and a collection with all actually downloaded files to get an actual characteristics of the examined domains.

From the previous example, we can derive the following list that represents a standard life cycle of each project:

1. Creation of a new project and configuration of repository, storages and crawlers,
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 clusters and collections,
6. Document classification and assignment into collections, and

[1]`http://dbpedia.org`

7. Computation of final statistical reports over particular collections.

Projects encapsulate inserted documents, configuration of analyses and computed data and represent a single research intent. During creation of a new project, the user can (besides other configuration) optionally select document types the project should be dedicated for. This selection is defined by a set of regular expression patterns over types.

The next step is a selection of *analyses* to be used in a given project (Figure 2.2). This selection is based on a list of all currently available *plugins*, but the user can only instantiate a particular plugin once, if such plugin is not configurable. As it will be explained later, plugins offer their functionality through methods. An integral part of configuration of analyses is also the definition of desired ordering of such methods.

Next, the user can import or download required documents. If a project involves type filtering, a given document is removed, if it does not match any of the provided patterns. The following step is creation of new clusters with sets of collections. Once a new collection is created, the classification of all existing documents satisfying other filtering criteria is automatically initiated. The final step is closing of clusters which invokes aggregation of results into reports. The reports are stored permanently and the user can browse them any time later.

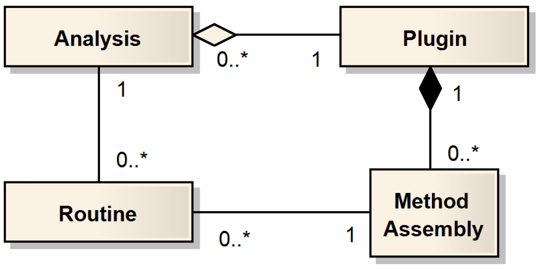


Figure 2.2: Analysis model diagram

Model of Documents Each document to be analyzed is characterized by a pair of physical and logical *resources* (Figure 2.4). The first one is the URL address from which the file was really imported, downloaded or sought from. The second one represents an address *Analyzer* "thinks" the original file should be located at. However, the guessing heuristic is not currently completely implemented and is a subject of our future work which will exploit and combine the crawling module (Section 2.4) and the query analysis (Section 2.7). *Analyzer* itself is able to maintain multiple versions of the same file. Several consecutive import or download *sessions* are always grouped together into *chains*, defining a relative *age* for all documents in them (Figure 2.3). It is not allowed to have more than one document of the same logical resource and the same relative age in a project.

The document entity itself is only an abstraction of a file – data *content* of a given file is treated independently and maintained using storages. Once a new document is inserted into a project, the corresponding file content is bound with this document entity. When generating content corrections, a new content version is always created and the previous one is thrown away, unless the previous one is the original one.

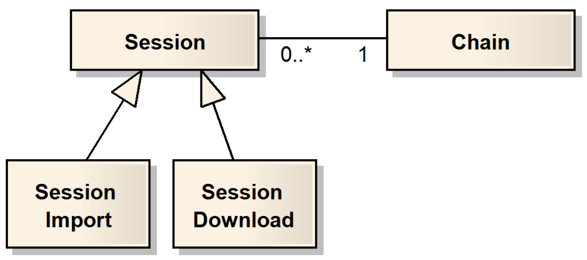


Figure 2.3: Session model diagram

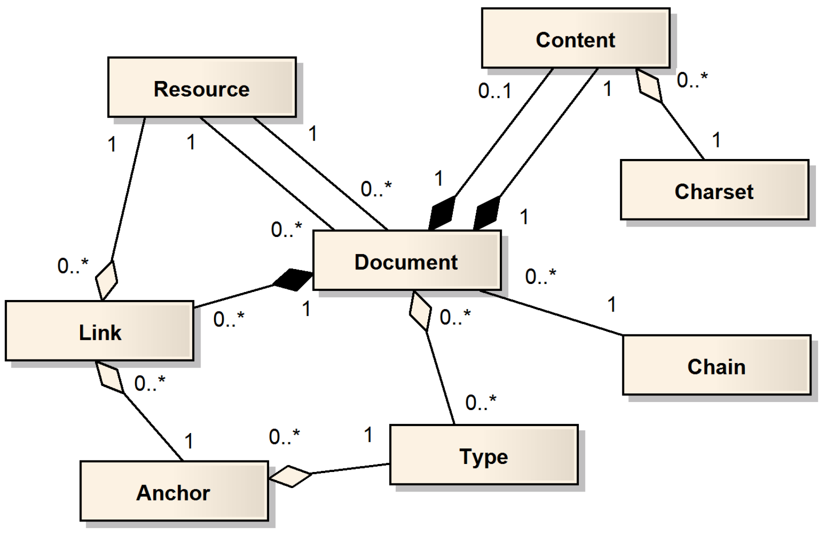


Figure 2.4: Document model diagram

During the first steps of document processing, document *types* are recognized. Because this detection is realized by plugins, the typing concept is not limited and developers are allowed to work with their own typing namespace. However, we have proposed to harness standardized MIME types. The most important fact about types is that each document is described by a set of recognized types, not only a single type. The idea behind types is simple: plugins recognize types, plugins analyze only documents of selected known types and, finally, projects can be restricted to processing of selected types only.

Analyzer also supports detection and processing of *links* between documents. A *link* is a typed reference from a source document to a target document, e.g., schema declarations in XML documents or image source files definitions in HTML [18] pages. The system is able to automatically delay processing of a given document until all required and accessible target documents are present in a project too.

Model of Collections After all documents with the same relative age are inserted into a project, the phase of computation of *results* is initiated. Each result is a small piece of information computed by a configured plugin over a particular document (Figure 2.6). It is assumed that results are not the goal of analyses, they are created in order to be aggregated over multiple documents later on, in a form of *reports* over *collections*.

Collections are introduced in order to allow grouping of documents, i.e., creating named sets of documents (Figure 2.5). The process of particular document classifi-

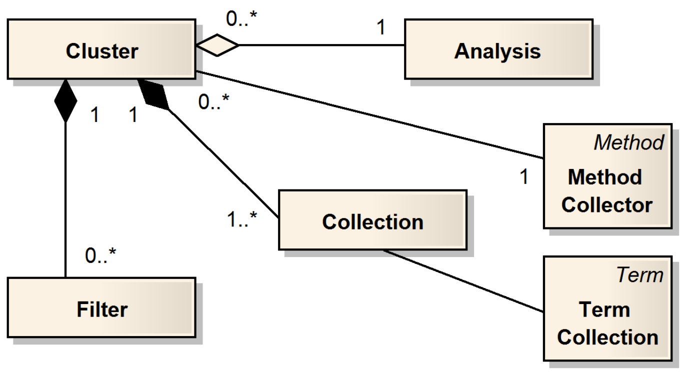


Figure 2.5: Collection model diagram

cation is, once again, semantically defined by a configured plugin itself. Despite this fact, the user is also able to filter documents using general criteria like, e.g., resource addresses or relative age restrictions. Since some collections can be mutually related (e.g., they classify documents into multiple categories using a shared set of criteria), *Analyzer* requires grouping of collections into *clusters*.

When the classification of all documents is done, computed results can be aggregated as previously outlined (Figure 2.6). This is done separately in each collection and, thus, the generated *reports* are always derived only from documents that are members of a given collection. However, not all documents may be provided with required results and, therefore, involved in this aggregation.

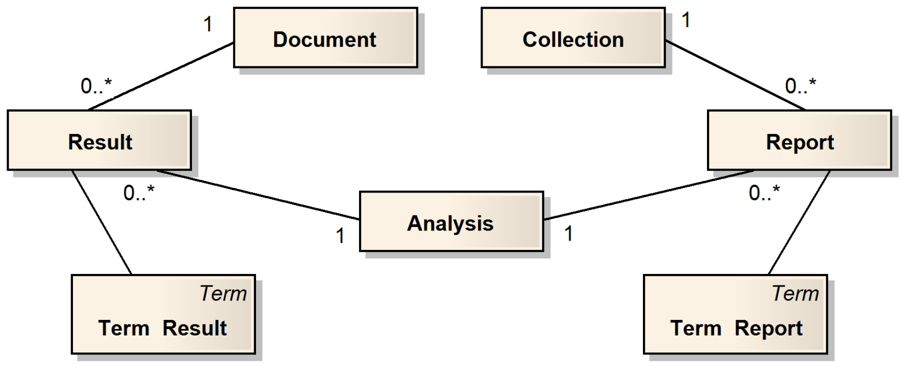


Figure 2.6: Result and Report model diagram

2.3.3 Model of 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. The current distribution of *Analyzer* includes a few basic plugins for processing general files and XML related files (see Sections 2.6 and 2.7), but the user is the one who is expected to create and use own extended plugins. It is also expected, but not required, that each plugin is determined for processing files of specific types. In other words, a plugin publishes its functionality and *Analyzer* makes it usable for analyses in projects.

Plugin Methods Disregarding the ability of a plugin to be configured (and thus, e.g., adjusted to particular analytical intents), each plugin specifies a set of document types that can be processed by it. This restriction is defined by regular expression patterns. The plugin functionality is provided through implemented *methods* (Figure 2.7). They are of eight predefined types listed in the following enumeration. Although these methods are not Java methods but classes, we can omit this fact for simplicity.

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

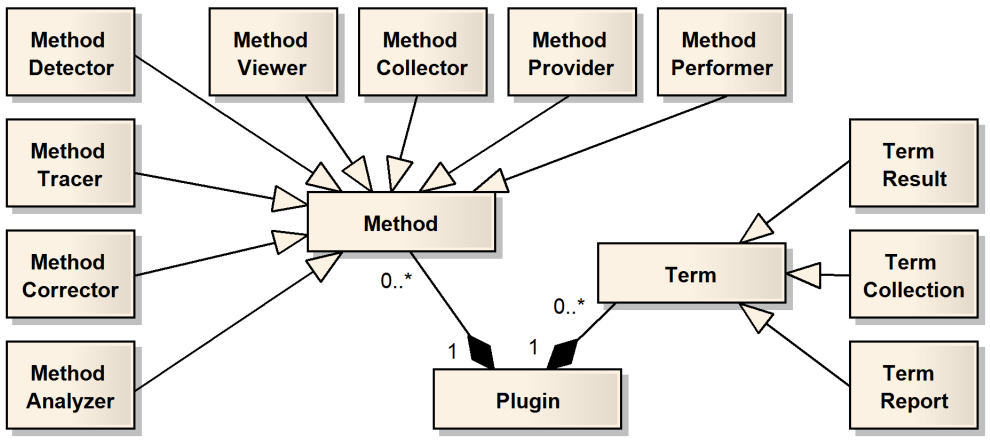


Figure 2.7: Plugin model diagram

After the user selects and configures all required analyses (plugins the user wants to use), the selection of particular available detectors, tracers, correctors and analyzers must be managed. This comprises not only the selection, but also the order of these methods. Despite different aims of these four method types, all of them may produce results. *Collectors* are methods that are responsible for classifying documents into collections. In other words, they make the decision, whether a given document belongs to a given related collection, or not. Once the user closes a cluster, *Analyzer* invokes *provider* methods over all its collections in order to aggregate results into reports. Finally, *viewer* and *performer* methods are used for presenting computed results over documents and computed reports over collections respectively.

Execution of Methods The execution of tasks representing plugin methods is similar to the execution of other tasks, *Analyzer* only wraps the code written by the plugin programmer, invokes the computation and handles potentially raised errors or other forms of incorrect processing.

All plugin methods share the way how they access the functionality of *Analyzer* and how they acquire data about documents or other entities they are processing or generating. These requests are processed by *mediators*, objects with well known interface and contract. Each method type works with own specialized mediator, which allows only for relevant operations.

The mediator itself in fact only pretends processing of all these requests, internally simulates required actions and the real execution is postponed until the very end of a given task execution. As a consequence, we are able to reveal several forms of inconsistent behavior based on the violation of a published plugin contract.

Implemented Plugins The implementation of *Analyzer* comes with a few created plugins, which are ready for use. If we omit sample plugins demonstrating only framework possibilities, there are three main groups of plugins: a universal plugin for basic analyses of documents regardless their types, a plugin for XML documents and their schema analyses (see Section 2.6) and, finally, a plugin for XQuery and XPath analyses (see Section 2.7).

2.3.4 Implementation of *Analyzer*

Analyzer is implemented in Java 6 language [90] as a desktop application with a robust GUI. It is built on top of NetBeans 6.8 platform [88] and capable of the cross-platform usage.

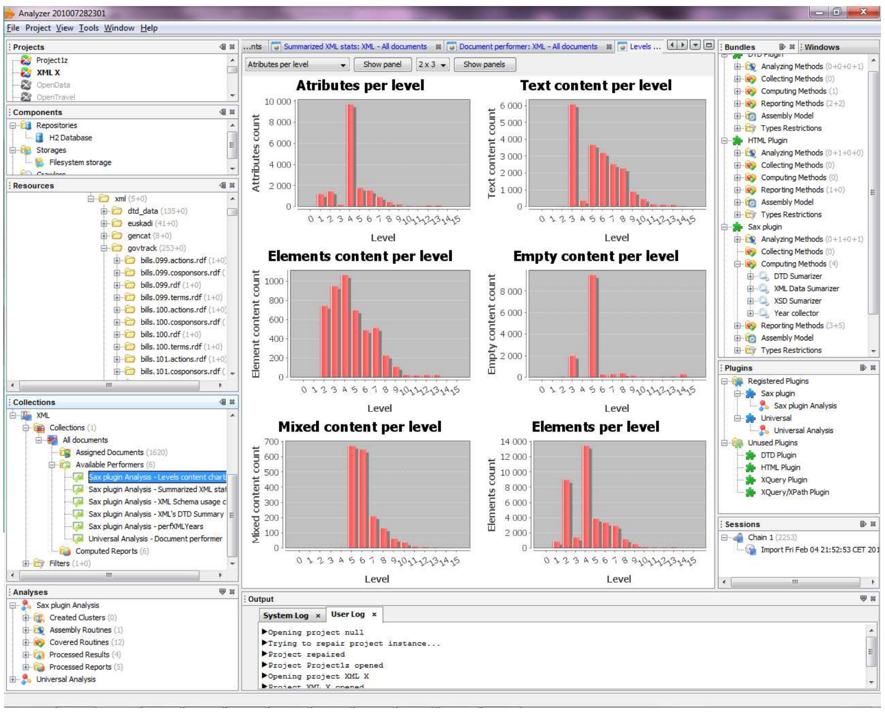
The GUI of *Analyzer* is based on the possibilities of the NetBeans platform. It brings the complete environment for creating and managing projects and performing analyses from their configuration to browsing of computed reports. A sample screenshot image can be found in Figure 2.8.

The default *Analyzer* distribution contains implementation of three repositories (MySQL server database through MySQL Connector 5.1.7 [91], embedded Apache Derby 10.5.1.1 database [92] and embedded H2 Database 1.1.117 [93]), two crawlers (simple built-in crawler and flooding Egothor 1.0 crawler [94]) and two storages (filesystem storage and dedicated Egothor storage).

2.3.5 Performance Experiments

Apparently, the key performance role is represented by repositories and, therefore, the main impact brings primarily the number of analyzed documents. We configured two simple analyses with four methods for generating results in total. The documents were inserted into a project using import without copying of physical files. Finally, a single cluster was created and all processed documents were classified into its two of six collections.

Table 2.1 shows results of performed experiments over three different sets of documents. All tests were executed using a PC with Intel Core 2 Quad Q9550 2.83 GHz processor, 4 GB RAM and Gentoo Linux 10.1 operating system. The analyzed documents were stored on a local hard drive and also all three repositories stored their



```
▶Opening project null
▶Trying to repair project instance...
▶Project repaired
▶Project Project1z opened
▶Opening project XML X
▶Project XML X opened
```

Figure 2.8: Sample screenshot of *Analyzer*

Set	Document count & size	Repository database	Document import	Results computation	Collections filling	Reports computation
A	1,000 x 100 kB	Derby	7 s	60 s	14 s	12 s
		H2 DB	2 s	12 s	6 s	1 s
		MySQL	3 s	19 s	9 s	< 1 s
B	10,000 x 10 kB	Derby	45 s	13 min	5 min	11 min
		H2 DB	10 s	100 s	90 s	60 s
		MySQL	15 s	135 s	70 s	10 s
C	100,000 x 1kB	H2 DB	1 min	150 min	150 min	16 h
		MySQL	3 min	22 min	14 min	1 min

Table 2.1: Performance characteristics

internal data locally on the same drive. MySQL Community Server 5.0.84 was installed locally and used through JDBC connector 3.0. Both filling of collections and computation of reports phases are based only on querying of repository (document contents are never read).

It is worth noting that the purpose of these experiments was to show capabilities of the framework itself and not particular plugins, e.g., for XML documents analyses. Therefore, we used special plugins with methods of constant time complexity only. As we can see, there is a significant difference especially between H2 and MySQL. This can be explained by the inability of H2 to work with defined auxiliary index structures during selection queries.

As we have mentioned, there are four key aspects of the analytical process performed by *Analyzer* – data crawling, data correction, structural analysis and query analysis. In the following four sections we discuss them in detail. For each of the four topics we briefly describe the state of the art of the respective area and then we provide a description of the particular decisions and especially contributions applied in *Analyzer*.

2.4 Data Crawling

The first key aspect of every data analysis is data gathering. As we have mentioned, *Analyzer* supports several types of input of the analyzed data, whereas the interesting one is the possibility of data crawling. The development of XML is closely tied to the Web; therefore, the Web is expected to contain vast amounts of XML-related data. Nevertheless, collecting the data is surprisingly difficult. While there is a number of crawlers used to collect data from the Web, most of them are limited to HTML and widespread text-document formats like PDF [95].

XML-related data, which we consider in the broadest sense in this section, include all XML-based formats (including, e.g., XML Schema and XSLT) and related non-XML languages (like DTD or XQuery). When crawling the Web, some documents of these types may be found linked from HTML pages; however, others are referenced from the *primary documents*, like their schemas or included documents. Thus, the *secondary documents* cannot be located using HTML-based crawling – instead an XML-aware crawler must be able to parse both XML documents and the related formats (DTD etc.) to extract the links to the secondary documents.

Crawling the Web correctly is a difficult task, entangled within performance bottlenecks and surrounded by ethics and copyright rules. Creating a new crawler from the scratch is apparently a senseless effort. Thus, we have opted to adapt and extend an existing HTML-oriented crawler. For our purposes we have evaluated the following systems:

- *Xyleme/Larbin* [96, 97] was included in our list due to its ability to handle XML-to-DTD links. However, since Xyleme is not an open-source software, it might not be extended in our project.

- *Egothor* [94] was a system developed at the Charles University in Prague and, therefore, it was the first candidate for extension. Unfortunately, the community of Egothor developers is too small to ensure the required long life of the system.

- *Apache Nutch* [98] is an open-source project from the Apache family. Thanks to its system of *extension points*, it originally seemed that the extension for XML-related data might be completely implemented using plugins.

- *Bixo* [99] is a *topical crawler* focused on mining data from selected locations, thus using a strategy different from traditional crawlers. However, it is tightly coupled with the mining methods.

- *Google* and Google Web APIs [100] is included in our list as a kind of a benchmark because Google allows the to search specific file types (like DTD) in its huge collection. Unfortunately, the exact method which Google uses for locating the specific documents is unknown.

As the previous list suggests, we selected the Apache Nutch system. We extended the system with a number of plugins and tuned its configuration towards the search for XML-related data. In addition, we had to modify the source code of Nutch at few places. Our modifications and extensions together realized the following alterations to the original behavior of the crawler:

- Improved address filtration – avoiding unwanted protocols (mailto, javascript) and formats (PDF, MP3 etc.).
- Altered document filtration – cutting-out excessively large HTML documents, assuming that they would unlikely contain any XML-related links.
- *Whitelisting* apparently XML-related documents based on their reported MIME-type and (a part of) contents.
- *Blacklisting* unwanted documents. (Due to widespread errors in the Web content, we found blacklisting more efficient than whitelisting.)
- Altered scoring mechanism – favorizing XML-related data in the download queue.
- Parsing XML-based documents and locating external references in them.

Our XML parser, based on a SAX implementation from the Xerces [101] family, is used to locate the following kinds of links in XML-based documents:

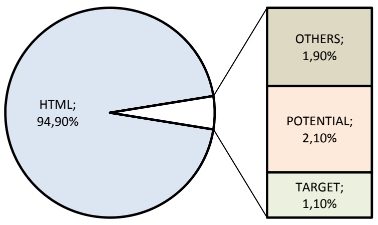
- `schemaLocation` and `noNamespaceSchemaLocation` attributes in any XML document,
- `import`, `include`, and `redefine` elements in XSDs,
- `import` and `include` elements in XSLT programs,
- *processing instructions* in XML Style Sheets, and
- `include` and `externalRef` elements in RELAX NG [102].

Another important fact is that many Web documents are malformed but still usable in crawling. Therefore, we made our parser robust with respect to non-well-formed data.

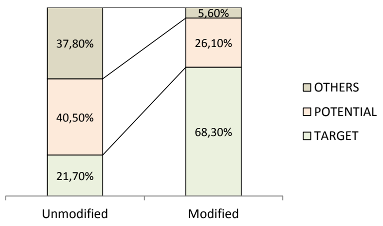
To depict the features of our modifications Figure 2.9(a) shows the percentage of document types encountered and downloaded during a testing run of unmodified Nutch system. The small non-XML part of these data is shown in detail in Figure 2.9(b) – the left column corresponds to the behavior of the original Nutch system, the right column displays the results of the system after our modification. The figures are based on medium-scale tests – approximately 1 million documents of which 3.62% were XML-related.

2.5 Processing of Incorrect Data

Documents gathered by automatic crawler as described in the previous section, or imported manually from different sources (e.g., filesystem), can contain several types of structural or semantic errors, which has to be solved. In *Analyzer*, we include error processing as the optional part of document processing. During this phase, *corrector*



(a) Types of crawled documents



(b) Types of crawled documents after improvements

Figure 2.9: Results of experiments with data crawler

methods of available plugins are able to modify data contents of the documents, or discard the document from analyzed set. Therefore, a plugin developer may propose methods for document correction and, thus, other methods can be assured they are working only with correct data.

In the scope of XML documents, which we aim at, we can witness rather surprisingly high number of documents involving various forms of errors [14]. These errors can cause that documents are not well-formed, they do not conform to the required structure or have inconsistencies in data values. Anyway, the presence of errors causes at least obstructions and may completely prevent successful processing. Generally, we can modify existing algorithms to deal with errors, or we can attempt to modify invalid documents themselves.

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 or XML Schema, i.e., a *regular tree grammar* with the expressive power at the level of *single-type tree grammars* [81]. Having a potentially invalid XML document, we process it from its root node towards leaves and propose minimum corrections of elements in order to achieve a valid document close to the original one. In each node of a tree we attempt to statically investigate all suitable sequences of its child nodes with respect to a content model and once we detect a local invalidity, we propose modifications based on operations capable to insert new minimum subtrees, delete existing ones or recursively repair them.

The remaining parts of this section 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 [76, 103].

2.5.1 Existing Approaches

The proposed correction model is based primarily on ideas from [104] and [105]. Authors of the former paper dynamically inspect the state space of a finite automaton for recognizing regular expressions in order to find valid sequences of child nodes with minimum 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 document querying, but not repairing.

Next, we can mention an approximate validation and correction approach [106] 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 [107, 108].

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 [104, 106] are not able to find repairs of more damaged documents, we are able to always find all minimum 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 [109] and performed experiments show a linear time complexity depending on a number of nodes in documents.

2.5.2 Proposed Solution

Our correction framework is capable to generate local structural repairs for locally invalid elements. These repairs are motivated by the classic Levenshtein metric [110] 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 ordinary strings, but sequences of nodes, which, in fact, are not only labels, but also entire 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 minimum repairs, i.e., repairs with the minimum distance to the grammar and the original data tree according to the introduced cost function.

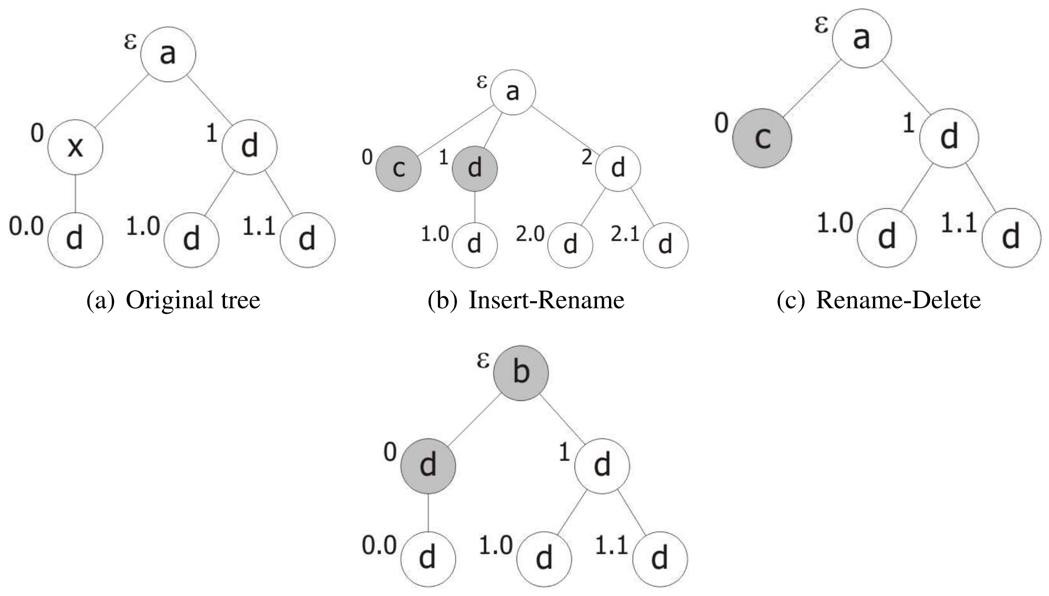
In order to illustrate the correction process throughout the following paragraphs, we will use a sample XML document based on a fragment:

<a><x><d/></x><d><d/><d/></d></a>.

The derived data tree $\mathcal{T}$ is depicted in Figure 2.10(a). Its underlying tree has nodes $\{\epsilon, 0, 0.0, 1, 1.0, 1.1\}$ and element labels are inscribed in nodes.

We want this document to conform to a simple local tree grammar $\mathcal{G}$, which requires that the label of the root node can only be a or b, child nodes of element a should match a regular expression $c.d^*$, elements b and d may contain an unlimited number of elements d, and, finally, element c should always be empty. Obviously, the sample data tree $\mathcal{T}$ is not valid against $\mathcal{G}$, since element x is not allowed by the grammar at all.

Edit Operations Edit operations are elementary transformations that are used for altering invalid data trees into valid ones. They behave as deterministically defined



(a) Original tree (b) Insert-Rename (c) Rename-Delete

(d) Rename-Rename

Figure 2.10: Sample XML tree with its three repairs

functions, performing small local modifications with a provided data tree. Though 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 the following edit operations:

- insert a new leaf node,
- delete an existing leaf node,
- rename a label of a node,
- push a group of adjacent sibling nodes lower under a newly inserted internal node, and
- pull all sibling nodes one level higher deleting their original parent node.

Moreover, we have also formally studied other node and attribute operations. However, these operations were not yet fully implemented and, therefore, will be omitted in the rest of this section.

Update Operations Edit operations can be composed together into sequences. And if these sequences fulfil certain qualities, they can be classified as *update operations*. We have proposed

- insertion of a new minimal subtree,
- deletion of an existing subtree, and
- a recursive repair of a subtree with an option of changing a label of its root node.

Anyway, the purpose of each update operation is to correct a local part of a data tree in order to achieve its local validity. Unfortunately the correction algorithm does not generate these operations. The algorithm generates repairs based on repairing instructions, which are subsequently translated into sequences of edit operations. And this is the reason, why update operations are not defined deterministically similarly to edit operations. Having a particular sequence of edit operations, we can inspect its subsequences and if all required conditions are satisfied, a given subsequence can be viewed as an update operation of a corresponding type.

Assume that we have edit sequences

- $\mathcal{X}_1 = \langle addLeaf(0, c), renameLabel(1, d)\rangle$,
- $\mathcal{X}_2 = \langle renameLabel(0, c), removeLeaf(0.0)\rangle$ and
- $\mathcal{X}_3 = \langle renameLabel(\epsilon, b), renameLabel(0, d)\rangle$.

Applying these sequences separately to data tree $\mathcal{T}$ from our example, we obtain data trees depicted in Figures 2.10(b), 2.10(c) and 2.10(d) respectively. If we use a unit cost function, these three data trees represent all minimal repairs of the original tree with the cost 2.

Repairing Instructions Assume that we are in a particular node in a data tree and our goal is to locally correct this node, which, passing over attributes, especially involves the correction of 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 minimum combinations of local optimums, meaning minimum repairs for all subtrees of original child nodes of the inspected one.

However, we need to find all minimum repairs, and since edit operations require particular positions in a current data tree to be specified, we cannot use them to describe all repairs. Assume, for example, that we have several options how to correct the first child node. If we delete it, all positions of nodes to the right must be shifted by one to the left, but if we accept the first child node, the original positions preserve. Thus, we are not able to use edit operations for describing multiple different repairing sequences.

The problem with the continuously changing numbers of positions is solved by the model of repairing instructions. We have exactly one instruction for each edit operation and these instructions represent the same transforming ideas, however, do not include particular positions to be applied on. Having a particular sequence of repairing instructions, we can easily translate it into the corresponding sequence 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 these involved nodes. As already outlined, these actions follow the model for measuring distances between ordinary strings. The Levenshtein metric is defined as the minimum number of required elementary operations to transform one string into another. These operations are insertion of one new symbol, deletion of an existing one and also replacement of an existing symbol with a new one. We follow the same model, however, we have edit and update operations respectively and sequences of nodes. The given sequence can be viewed as an ordinary string over labels of its nodes. For example, insertion of a new subtree at a given position stands for insertion of its label into the corresponding string of labels and, of course, recursive processing 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 attempt to inspect all these words statically using a notion of a correction and derived multigraphs.

Anyway, suppose that the algorithm has already processed first few nodes from the inspected sequence of sibling nodes, thus all nodes from the corresponding prefix of the original sequence are already involved in corrections. Now, the algorithm must consider all possible actions that can be selected in order to involve at least one next node from the original sequence. The possibilities are modeled using the notion of correction intents.

In other words, the correction algorithm in each parent node has a variety of options how to achieve its validity and particular steps performed with its child nodes are called *correction intents*, because we always examine one possible action from more permitted ones.

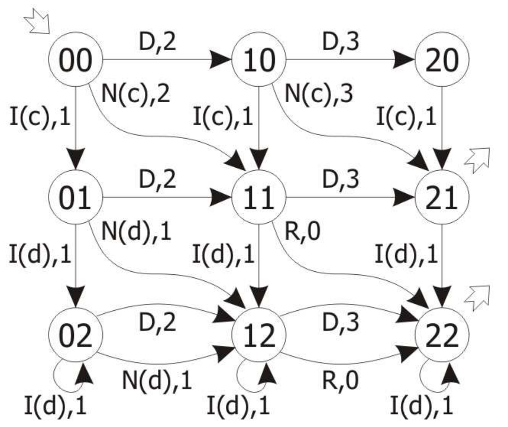
Correction Multigraphs All existing correction intents in a context of a given node can be modelled using a correction multigraph. Suppose that we need to process a sequence of child nodes with n nodes. This means that the graph will have $n+1$ strata, numbered from 0 to n. Being on a stratum with number i, we have already processed right i first nodes from this sequence.

Each stratum is constructed from the Glushkov automaton for recognising the provided regular expression restricting the given sequence. This means that there are vertices corresponding to states of the automaton and directed edges reflecting the transition function in each stratum. Each such edge represents a new tree insertion operation. Similarly, we can define edges between strata to represent other allowed operations.

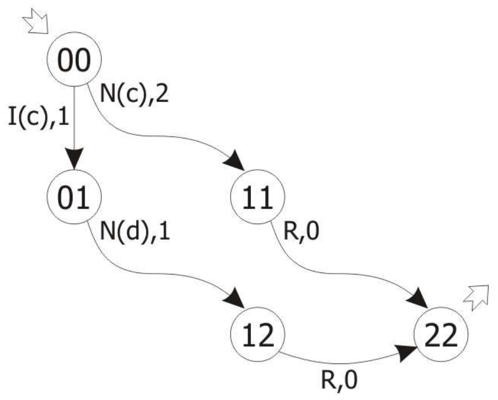
In other words, the correction multigraph represents all correction intents that can be derived for this sequence. And more precisely, each intent is represented by some edge in this multigraph. However, there can be intents that are represented by more edges at once.

In order to find best repairs for a provided sequence of nodes, we need to find all shortest paths in this multigraph, assuming that every edge is rated with an overall cost of corresponding nested correction intent associated with such edge. However, to resolve these costs, we need to fully evaluate associated intents. 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.

If we return to our sample data tree $\mathcal{T}$, we can represent all nested correction intents derived for a root node a and a sequence $\langle x, d \rangle$ of its child nodes with respect to a regular expression $c.d^*$ by a correction multigraph in Figure 2.11(a). For simplicity, edges are described only by abbreviated intent types (I for `insert`, D for `delete`, R for `repair` and N for `rename`), supplemented by a repairing instruction parameter if relevant and, finally, the complete *cost* of assigned intent repair. Names of vertices are concatenations of a stratum number and an automaton state.



(a) Sample correction multigraph



(b) Sample repairing multigraph

Our goal is to find all shortest paths from the source vertex 00 to any of the target vertices 21 or 22 in the last stratum. Having found them, we can represent them in a form of the repairing multigraph in Figure 2.11(b).

Repairs Construction Each correction intent can essentially be viewed as an assignment to the nested recursive processing. This model, in fact, has a transparent relation with a structure of an underlying tree itself and its processing from the root node towards leaves. The entire correction of a provided data tree is initiated as a special starting correction intent for 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, which involves gathering of found repairs. This means that after we have found the desired shortest paths at a given level, we encapsulate them in a form of a compact repair structure and pass it one level up, towards the starting correction intent.

Having found the shortest paths in the repairing multigraph for the starting intent, we have found repairs 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 minimum distance.

Correction Algorithms Now we have completely outlined the model of the proposed correction framework. 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. They both produce the same repairs, but there are key differences in their efficiency.

The first algorithm is able to directly search for the 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 executed by a simple scheduler.

2.6 Structural Analysis

Having crawled and corrected the data, we can start with their analysis. In this section, we discuss several metrics, we implemented in current version of *Analyzer*. We have to note that the implemented methods cover only basic characteristics of XML, DTD and XML Schema, and they are based on previously published statistics of XML formats (see Section 2.8).

The implemented methods cover XML documents and the related schemas expressed in DTD and XML Schema. We created one plugin for each of the document types and one universal plugin to determine basic file attributes. The plugin for XML documents is applied to XML Schema documents, but beside the basic statistics, the usage of some specific constructs is measured.

We expected that the analyzed documents can be very large, so we based the implementation of plugins on *Simple API for XML* (SAX) [111]. SAX allows for efficient work with large files, but, on the other hand, it complicates some analytic methods, e.g., XPath fragment search.

2.6.1 Common Properties

Although, we focused on three different formats, they have many common aspects which can be analyzed in a similar way. Firstly, we analyze the number of entities used in the document, e.g., elements, attributes, declarations, etc. Secondly, we analyze the structural complexity of the used model. To define the structural characteristics of the document, we first need to define the XML document and XML schema. We will use the definitions as presented in [3] and [25]. The XML documents are expressed as ordered trees and XML schemas are expressed as regular expressions over element names.

Definition 1. *An **XML document** is a finite ordered tree $T = (\Sigma, N, E, r)$, where Σ is a finite alphabet, N is a set of nodes of the tree, E is a set of edges of the tree, and $r \in N$ denotes a **root element** of the tree. Each node $\in N$ is associated with a **type of the node** which can be one the following: **element**, **attribute**, **text**, **processing instruction**, or **comment**. Nodes with element or attribute type are also associated with a node label $l \in \Sigma$ called an **element name** or an **attribute name** respectively. The tree T is called Σ-tree*

Definition 2. *A **DTD** is a collection of element declarations of the form $e \rightarrow \alpha$, where $e \in \Sigma$ is an element name and α is its content model, i.e., regular expression over Σ. The content model α is defined as $\alpha = \epsilon \mid pcdata \mid f \mid (\alpha_1, \alpha_2, ..., \alpha_n) \mid (\alpha_1 | \alpha_2 | ... | \alpha_n) \mid \beta* \mid \beta+ \mid \beta?$, where ϵ denotes the empty content model, pcdata denotes the text content, f denotes a single element name, "," and "|" stand for concatenation and union (of content models $\alpha_1, \alpha_2, ..., \alpha_n$), and "*", "+", and "?" stand for zero or more, one or more, and optional occurrence(s) (of content model β) respectively.*

*One of the element names $s \in \Sigma$ is called a **start symbol**.*

In the following definitions we will focus on the key characteristics of XML data. Due to space limitations and similarity, we will provide definitions for XML schemas, in particular DTDs and assume that their modifications for XSDs and XML documents are simple and apparent.

The complexity of the content model can be described by its depth.

Definition 3. *A **depth** of a content model α is inductively defined as follows:*

- *depth(ϵ) = 0;*
- *depth(pcdata) = depth(f) = 1;*
- *depth(α_1, α_2, ..., α_n) = depth(α_1| α_2| ...| α_n) = max(depth(α_i)) + 1; 1 $\leq$ i $\leq$ n*
- *depth(β*) = depth(β+) = depth(β?) = depth(β) + 1.*

Another important characteristic of the structure of whole schemas/documents are fan-out and fan-in.

Definition 4. *A **fan-out** of an element e is the cardinality of the set { f | e' is the element name of element e, e' $\rightarrow$ α and the element name f' of element f occurs in α }.*

Definition 5. *A **fan-in** of an element e is the cardinality of the set { f | f $\rightarrow$ α' and the element name e' of element e occurs in α' }.*

Other metric of complexity of both documents and schemas is the usage of different types of content of an element. Generally, we can distinguish three types of content: empty, text, element and mixed. The former two are trivial; the latter two are defined as follows.

Definition 6. *A content model α is **element**, if $\exists e \in \Sigma$, such that e occurs in α.*

Definition 7. *A content model α is **mixed**, if α = (α_1|...|α_n|pcdata)*|(α_1|...|α_n|pcdata)+, where n $\geq$ 1 and $\forall$ i, such that 1 $\leq$ i $\leq$ n, content model $\alpha_i \neq \epsilon \wedge \alpha_i \neq$ pcdata. An element e is called **mixed-content** element if its content model α is mixed.*

Last but not least important characteristic is the usage of specific structures. As mentioned in Section 2.2, a controversial characteristic are recursive elements.

Definition 8. *An element e is **recursive** if there exists at least one element d in the same document, such that d is a descendant of e and d has the same element name as e.*

2.6.2 XML Documents

We created plugins to measure the following properties on XML documents:

- The size of the XML document, e.g., in bytes, the number of elements or the number of attributes
- Maximum depth of the document
- Distribution of various types of content model over different levels
- Recursion of elements

- Maximum and average fanout
- Usage of XML Schema versus DTD
- Distinct element/attribute name usage
- Namespace usage

2.6.3 DTD Analysis

For the DTDs, the system contains methods to compute the following statistics:

- The size of the DTD, e.g., number of declarations of elements, attributes, notations, entities etc.
- Number of DTD specific declarations of elements content type (`empty`, `any`, "`|`", "`,`")
- Number of DTD specific declarations of attribute optionality (`#REQUIRED`, `#IMPLIED`, `#FIXED`)
- Usage of keys (i.e., attribute data types `ID` and `IDREFS(S)`)
- Maximum, minimum and average depth
- Average and maximum fan-outs and fan-ins

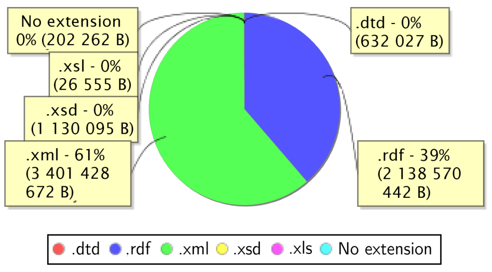
2.6.4 XML Schema Analysis

The complexity of XSDs is basically measured in the same way as XML documents, since each XSD is at the same time an XML document. Beside these properties, we created a plugin for measurement of the usage of specific constructs as follows:

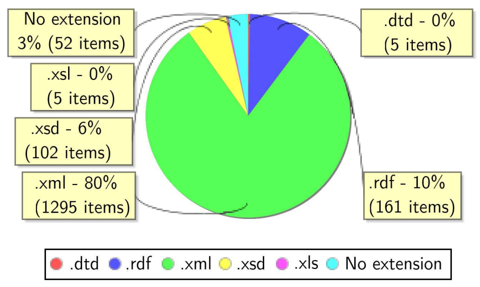
- Type specification (`simpleType` and `complexType`)
- Restriction and extension of existing types
- Content model of the elements (`sequence`, `choice`, `all`)
- Element groups and attribute groups (`group`, `attributeGroup`)

2.6.5 Results

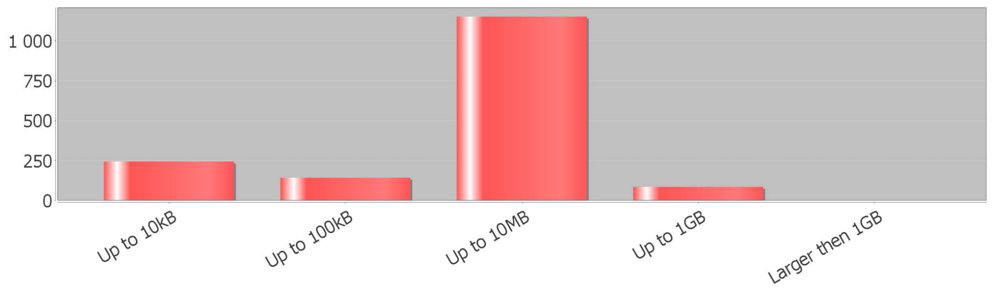
Due to space limitations, we prepared a small sample of data to show the capabilities of the framework. A complete throughout analysis, together with analysis of related operations (see Section 2.7) will be a subject of our very next future work and a separate paper. We used the current implementation of *Analyzer* with the basic implemented set of plugins. We run the application on a common dual-core processor with 2 GB RAM. The projects was created with the H2 DB repository and the filesystem storage.



(c) Size distribution of types



(d) Type distribution



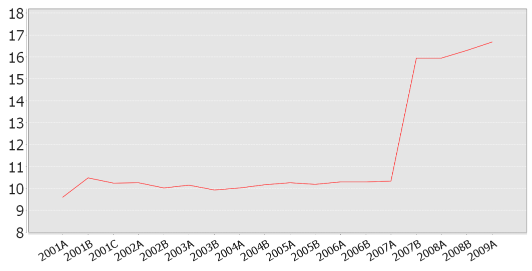
(e) Size distribution

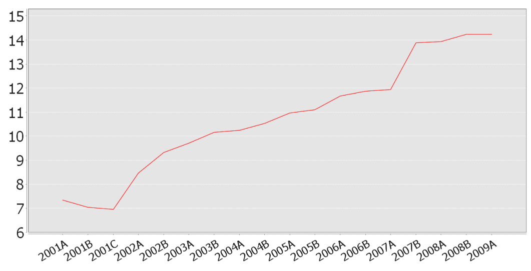
Figure 2.11: Document distribution

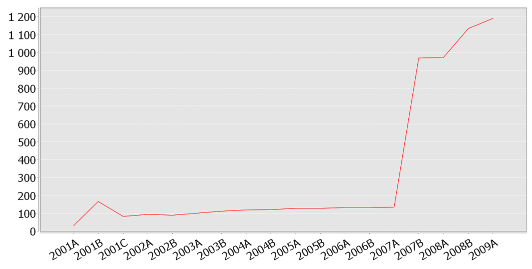
Data Sources For the experiments we used several data sources to get a variable sample of the real-world data. As the open data became more popular, the methods for their processing are more required. Despite widespread usage of XML, we can expect that the part of the data will be simple conversions from other formats like XLS[2] or CSV[3], so we used also a small sample of public available data to get some basic information about their structure. Firstly, the XML documents are gathered mostly from the U.S. federal executive branch data sets on `data.gov`. Some data was downloaded from open data server of *the Government of Catalonia*[4] and the rest of the documents from *U.S. congress*[5] and *Open Data Euskadi*[6]. The data sets generally contain financial reports or geographical information of a government related information. Secondly, we used the *OpenTravel* specification[7] as a sample of XSD documents and compared their versions over last 9 years.

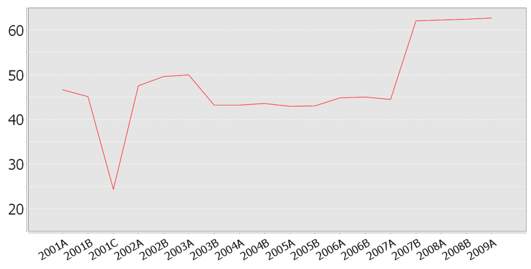
XML Documents Statistics In the first phase we took all data from all open data sources and computed global statistics over them. The results are show in Figure 2.11. The document type distribution by size is depicted in Figure 2.11(c) and the type distribution by number in Figure 2.11(d). Since we gathered the XML documents, as we can see they are the main part of the data. On the other hand, the chart shows big average size of RDF documents. Only 161 RDF documents takes almost 40% of the total size. The distribution by size is illustrated in Figure 2.11(e) and shows that the majority of this sample are documents between 10kB and 10MB.

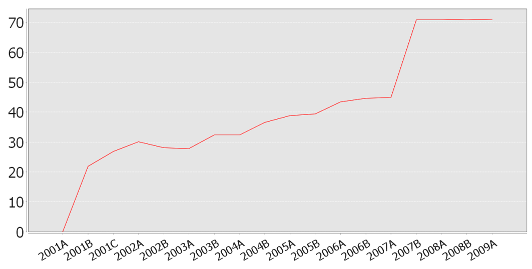
[2]Microsoft Excel format
[3]Comma Separated Values
[4]`http://opendata.gencat.cat/en/dades-obertes.html`
[5]`http://www.govtrack.us/data/rdf/`
[6]`http://opendata.euskadi.net/`
[7]`http://opentravel.org/`


(a) Distinct element names


(b) Average depth


(c) Average number of elements


(d) Usage of sequences


(e) Usage of extensions

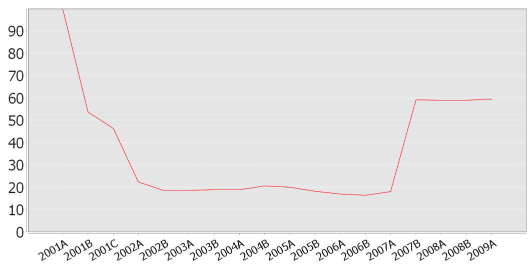

(f) Usage of simple types

Figure 2.12: Results of XSD analysis

Source	Total size	Maximum size	Average size	Document count
open.gov	2.5 GB	67 MB	2.3 MB	1156
gencat.cat	58.3 MB	40 MB	7.4 MB	8
govtrack.us	1.93 GB	77 MB	12.8 MB	253
euskadi.net	3.4 MB	1.7 MB	27 kB	124
All	4.5 GB	77 MB	2956 kB	1541
Source	**Maximum depth**	**Average depth**	**Used DTD**	**Used XSD**
open.gov	15	10.2	0.2 %	88.6 %
gencat.cat	11	6	0 %	0 %
govtrack.us	5	4.8	0 %	0 %
euskadi.net	9	6.2	41.93 %	0 %
All	15	8.87	7.4 %	63.3 %

Table 2.2: Results of XML document analysis

To get a more precise picture of the examined sample, we computed the basic statistics over the documents. We focused on the number of the used elements and attributes, depth and used schemas. The basic attributes of the examined files divided by the source are depicted in Table 2.2. The results show that the documents are generally flat, the average depth of `open.gov` sample exceeded depth of 10 with maximum depth 15. We can also see that XSDs are used only in the sample from `open.gov`. DTDs are partially used in documents from `euskadi.net`.

open.gov gencat.cat govtrack.us euskadi.net

According to the results we can say that a typical open data document is shallow and its size is up to 5 MB. Anyway, due to the sample size and the limitations of space we cannot make a general conclusion. We will focus on a detailed analysis over various large data sets in our future work.

XML Schema Statistics In the second experiment we focused on XSDs used in the OpenTravel specification. The data set contains 4,637 documents with total size of 194MB. The data set consist of several versions of the specification. In our experiment, we tried to show the evolution of the specification in terms of the size of the documents and the complexity of used constructs. The results of the analysis are shown in Figure 2.12. In particular, we focused on global characteristics like the number of distinct element names (see Figure 2.12(a)), the average depth (see Figure 2.12(b)), or the average number of elements (see Figure 2.12(c)). The second group of the examined values are XSD specific keywords. As an example, we show the evolution of the usage of sequences (see Figure 2.12(d)), extensions (see Figure 2.12(e)) and simple types (see Figure 2.12(f)).

According to the results, we can see that, quite naturally, the specification is getting more complex and its depth is increasing. The largest change came between versions 2007A and 2007B. On the other hand, the usage of XSD constructs is stagnating, only the usage of `extension` keyword is rising which reflects the general strategy of preserving backward compatibility.

2.7 Query Analysis

Besides XML data and/or schemas, whose analysis was described in the previous section, analysis of queries over XML data may also give some insight into the way how XML is used. Such analysis will be useful for implementers of query engines or storage systems; in addition, the queries may also reveal useful facts on the data themselves (e.g., the presence of recursion). There is a number of languages designed for processing and/or generating XML data. While the scientific community focuses on the XML Query (XQuery) language [21], real-world applications often use the XSLT [69]. These two languages, supported by their W3C standards, are currently the most widely used ones, although challenged by a number of more or less exotic languages like XDuce [112] etc.

XQuery 1.0 and XSLT 2.0 are powerful, Turing-complete [27] languages; however, their applications usually solve relatively simple problems like generating HTML pages or transforming XML between two schemas. Consequently, it is often believed that most applications use only small subsets of these languages. This observation is also supported by the fact that the most popular textbooks on XQuery or XSLT do not cover the languages exhaustively.

From the perspective of the implementor of an XQuery or XSLT processor, this observation suggests that a number of language features is rarely used and, therefore, not worthy of aggressive optimization. For example, the *following/preceding* axes [28] are used significantly less frequently than the *child/descendant* axes; consequently, the majority of indexing and querying techniques like twig joins [29] are limited to the *child/descendant* axes.

Note that we introduced the observation with the clause "it is often believed"; indeed, it was probably never confirmed by any statistically significant study. Such a study was among our goals in this project. However, some of the tools required by this study showed at least as interesting as the study itself. So, due to space limitations, similarly to the case of structural analysis we have decided to involve only an example of the query analysis, leaving a throughout version to the future work and a separate paper.

In this section, we describe *XQAnalyzer* – a tool designed to support studies that include analysis of a collection of XQuery programs. Since it is a novel and unique part of the framework assumed to be widely used by the researchers, it can be used both as a standalone application and a plugin of *Analyzer*. *XQAnalyzer* consumes a set of XQuery programs, converts them into a kind of intermediate code, and stores this *internal representation* in a repository (see Section 2.7.2). Subsequently, various analytical queries (see Section 2.7.1) 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.

The architecture of the *XQAnalyzer* is shown in Figure 2.13. 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.

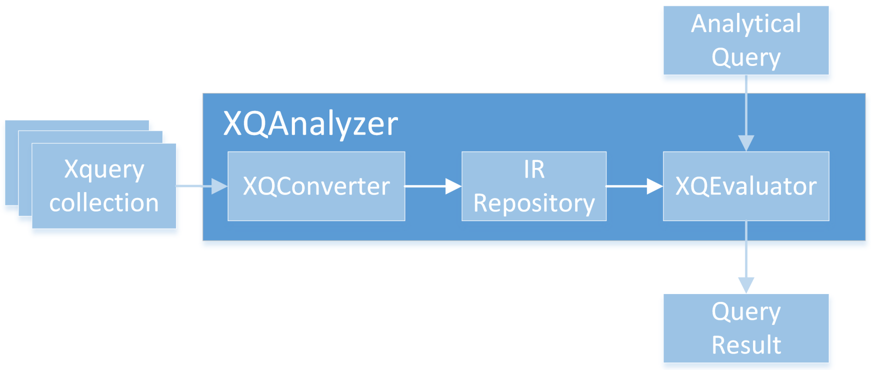


Figure 2.13: The architecture of the *XQAnalyzer*

2.7.1 Analytical Queries

In the *XQAnalyzer*, the term *analytical query* denotes a pattern or a condition placed on an 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.

The language in which analytical queries are placed shall be powerful enough to allow sophisticated patterns like “call to a user-defined function placed inside a FLWOR statement whose arguments are independent of the FLWOR control variable”. At the same time, the potential users of the system must be able to learn the language quickly.

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.

With the choice of XPath, the only remaining question in the design of the query language is the structure of the tree representing an XQuery program and its mapping to XML. In this representation, an analytical query is just an XPath query over the XML representation of query programs. The XML representation is discussed in the following section.

2.7.2 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 for 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*. Since we already decided to use a tree-based representation queried through XPath, our freedom of choice is in the following issues:

- The depth of the analysis performed before generating the internal representation.

- The "information density" of the tree, i.e., the number and the degrees of nodes assigned to an individual language feature and, consequently, the size of the tree assigned to an input program.

- The representation of repeated components – either by recursion or by nodes with unlimited number of children.

- The names and attributes of nodes, including technical details regarding their mapping to the XML.

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

- The *XQuery Grammar* defined in [21] using Extended Backus-Naur Form [113].

- The *Normalized XQuery Core Grammar* defined in [114] (also using EBNF).

Note that the XQuery formal semantics [114] is defined in terms of static/dynamic evaluation rules that may be considered as a kind of internal representation too. However, their application in our analytical environment would be impractically difficult.

There is also a number of formalisms defined in scientific literature. Among them, algebraic systems like XAT [115] might be easily adapted for our tree-based internal representation. However, these systems are always skewed towards a particular evaluation strategy (usually relational) and their use for other strategies would be difficult. For the same reason, we did not try to use *twig patterns* [29] in our representation.

Finally, it was proven [116] that each XQuery program may be translated to XSLT 2.0 [26]. Since each XSLT program is technically an XML document, the result of the translation may be used as internal representation of the input XQuery program. Unfortunately, the conversion from XQuery to XSLT is not straightforward due to minor differences in the semantics of similar constructs (FLWOR vs. `<xsl:for>`). Furthermore, only a part of XSLT syntax is expressed in terms of XML; the rest is hidden as XPath expressions inside the text of some XML attributes.

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.

With respect to the depth of the analysis, the Normalized XQuery Core Grammar requires only parsing and normalization. In the canonical XQuery-processing chain, it would be followed by the optional static type analysis and the dynamic evaluation phase. Since static type analysis produces only additional information to augment the existing tree, it does not influence our selection of internal representation.

Of course, in real-world XQuery processors, normalization is followed by conversion to an algebra or other representation. As we have discussed above, these representations, if ever published, are hardly suitable for a strategy-independent tool.

The Normalized XQuery Core Grammar defines the *concrete* syntax of the XQuery Core. Therefore, it must define syntactic properties like priority and associativity of

```
<Path initial-step="root">
  <Step>
    <Axis abbreviated="true" direction="forward"
        kind="descendant-or-self">
      <KindTest kind="any-kind"/>
    </Axis>
  </Step>
  <Step>
    <Axis abbreviated="true" direction="forward" kind="child">
      <NameTest name="car"/>
    </Axis>
    <Predicates>
      <Operator class="comparison" name="equals"
          subclass="general">
        <Path initial-step="context">
          <Step>
            <Axis abbreviated="true" direction="forward"
                kind="attribute">
              <NameTest name="type"/>
            </Axis>
          </Step>
        </Path>
        <Literal type="string" value="SUV"/>
      </Operator>
    </Predicates>
  </Step>
</Path>
```

Figure 2.14: Internal representation of an XPath expression

operators. Consequently, derivation trees constructed from this grammar have long branches containing semantically useless levels.

For analytical purposes, a more abstract representation is required, in a form of an *abstract syntax tree* (AST) [117]. An AST is present in almost every compiler; however, the corresponding *abstract* syntax grammar is rarely published or even standardized. In our case, we need an abstract grammar as close to the Normalized XQuery Core Grammar as possible. Therefore, we decided to start with the Normalized XQuery Core Grammar and to remove a part of the non-terminals corresponding to semantically useless levels that served only to define concrete syntax, collapsing the surrounding rules together. In few cases, we renamed the remaining nonterminals to more appropriate names (like Operator). The final set of nonterminals is listed in Table 2.3. When our internal representation is presented in the form of an XML document, these nonterminals become *XML elements*.

The rest of the semantic information is enclosed in *XML attributes* attached to the 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).

For example, the XPath/XQuery expression

```
//car[@type="SUV"]
```

is normalized to the following XQuery Core expression

```
/descendant-or-self::*/child::car
    [attribute::type="SUV"]
```

and then converted to the internal representation shown (in the XML form) in Figure 2.14.

The original form before normalization is described using the attribute

```
abbreviated="true"
```

in the `Axis` elements. The internal representation then may be queried using XPath expression like

```
          //Step[Axis[@kind="child"] and
Predicates/Operator[@name="equals" and Path and Literal]]
```

which finds any child-axis step combined with a predicate based on equality between a path expression and a literal.

2.7.3 Results

Since there is no standardized collection of real-world XQuery programs yet (except for small benchmarks like XMark [118]), we have chosen two artificial collections associated to the W3C XQuery language specification: the *XQuery Use Cases* [30] and the *XQuery Test Suite* [31]. 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 14,869 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 Table 2.3 we show the frequency of core elements of the language, named according to the abstract grammar nonterminals derived from the Normalized XQuery Core Grammar. The percentages are defined by the number of occurrences 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. The 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 2.4 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.

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

Table 2.3: 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	—	—

Element	Use Cases	Test Suite
following	—	0.44%
parent	—	0.50%
ancestor	—	0.44%
preceding-sibling	—	0.42%
preceding	—	0.42%
ancestor-or-self	—	0.44%

Table 2.4: Axis usage

2.8 Related Work

As we have mentioned in the introduction, *Analyzer* is a quite a unique tool in the area of analyses of both XML data and general data types. Not to mention the area of query analyses, where there currently exists neither such a framework, not even results of a respective analysis.

Considering the area of XML data analysis, we can find several papers which involve the results of various types of statistical data analyses. However, in all the cases the respective tool (or a set of tools) is not available, so the analyses are neither extensible, nor repeatable. The papers analyze either the structure of DTDs, the structure of XSDs, the structure of XML documents (regardless their schema), or the structure of XML documents in comparison with XML schemas. The sample data usually differ, whereas, since the authors did not use an advanced crawler, the set is usually quite small and unnatural.

DTD Analyses For the first time an analysis of the structure of DTDs, in particular 12 real-world DTDs, probably occurred in paper [2] and it was further extended in papers [3] (60 DTDs) and [4] (2 DTDs). They focused especially on the number of (root) elements and attributes, the depth of content models, the usage of mixed content, `ID`s/`IDREF`s, and *attribute optionality* (i.e., `#IMPLIED`, `#REQUIRED`, and `#FIXED`), non-determinism and ambiguity. A side aim of the papers was a discussion of shortcomings of DTDs, since the XML Schema was only in the status of a preliminary working draft. The most important findings are that real-world content models are quite simple (the depth is always less than 10), the number of non-linear recursive elements is high (they occur in 58% of all DTDs), the number of shared elements is significant, and that `ID`s/`IDREF`s are not used frequently.

XML Schema Analyses With the arrival of XML Schema, as the extension of DTD, a natural question has arisen: Which of the extra features of XML Schema not allowed in DTD are used in practice? Paper [5] is trying to answer it using statistical analysis of real-world XML schemas, in particular 109 DTDs and 93 XSDs. The most exploited features seem to be restriction of simple types (found in 73% of schemas), extension of complex types (37%), and namespaces (22%). The first finding reflects the lack of types in DTD, the second one confirms the naturalness of object-oriented approach, whereas the last one probably results from mutual modular usage of XSDs. The other features are used minimally or are not used at all. The concluding finding is that 85% of XSDs define local-tree languages that can be defined by DTD as well. Paper [25], that also focuses directly on structural analysis of XSDs, defines 11 metrics of XSDs and two formulae that use the metrics to compute complexity and quality indices of XSDs. Unfortunately, there is only a single XSD example for which the statistics were computed.

XML Data Analyses Paper [6] (and its extension [119]) analyzes the structure of about 200,000 XML documents directly, regardless eventually existing schema. The statistics are divided into two groups – statistics about the XML Web (e.g., clustering of the source Web sites by zones and geographical regions, the number and volume of documents per zone, the number of DTD/XSD references etc.) and statistics about the XML documents (e.g., the size and depth, the amount of markup and mixed-content elements, fan-out, recursion etc.). The most interesting findings of the research are that the structural information always dominates the size of documents, both mixed-content elements (found in 72% of documents) and recursion (found in 15% of documents) are important, and that documents are quite shallow (they have always fewer than 8 levels in average).

A much simpler document analysis performed in paper [24] consists of two parts – a discussion of different techniques for XML processing and an analysis of real-world XML documents. The sample data consists of 601 XHTML [120] Web pages, 3 documents in the *DocBook* format[8], an XML version of Shakespeare's plays[9] (i.e., 37 XML documents with a common simple DTD) and documents from the *XML Data Repository* project[10]. The analyzed properties are the maximum depth, the average depth, the number of simple paths, and the number of unique simple paths; the results are similar to the previous cases.

XML Data vs. XML Schema Analyses The work initiated in the previously mentioned articles is taken up by probably the latest paper in this field [14]. It enhances the preceding analyses and defines several new constructs for describing the structure of XML data (e.g., so-called *DNA* or *relational patterns*) and analyzes XML documents together with their eventual DTDs/XSDs that were collected semi-automatically, i.e., with interference of human operator. The reason is that automatic crawling of XML documents generates a set of documents that are unnatural and often contain only trivial data which cause misleading results. The collected data consist of about 16,500 XML documents of more than 20GB in size divided into 133 collections, whereas only 7.4% have neither a DTD nor an XSD. Such a low ratio is probably caused by the semi-automatic gathering.

The data were divided into five categories – *data-centric*, *document-centric*, *exchange*, *report*, and *research*. The first two categories correspond to classical categories [121], the other three are introduced to enable finer division. The statistics described in the paper are also divided into several categories – *global* (e.g., number of various constructs), *level* (i.e., distribution of various constructs per level), *fan-out* (i.e., branching), *recursive* (i.e., types and complexity of recursion), *mixed-content* (i.e., types and complexity of mixed contents), *DNA* (i.e., types and complexity of DNA patterns), and *relational* (i.e., types and complexity of relational patterns). They were computed for each document category and, if possible, also for both XML documents and XML schemas and the results were compared.

Most interesting findings and conclusions for all categories of statistics are partly expectable (e.g., that tagging usually dominates the size of document or that mixed-content elements are used in 77% of document-centric documents) and partly similar to the previous results (e.g., that the average depth of XML documents is about 5). However, there are also some very interesting observations and conclusions. For example, recursion statistics show that despite the typical assumptions recursion occurs quite often, especially, in document-centric (43%) and exchange (64%) documents, 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.

[8]`http://www.docbook.org/`

[9]`http://www.ibiblio.org/xml/examples/shakespeare/`

[10]`http://www.cs.washington.edu/research/xmldatasets/`

As we can see, the performed analyses reflect the development of XML technologies and usage of XML data in various applications. The problem is, that all the papers are relatively old (the first paper is from 2001, the last one is from 2006), so the results are obsolete. At the same time, there occur new XML technologies, data types and applications, whose analysis would be very useful. And, a similar study would be even more useful in the area of XML operations.

2.9 Open Problems

Even though the current version of *Analyzer* is a fully functional system that can be applied on analyses of real-world XML data (as partially shown in Sections 2.6.5 and 2.7.3), there are naturally various open problems to be focussed on.

Advanced Crawling Apart from classical crawling strategies on the basis of URLs used in HTML or XHTML linking constructs, detection of file types using file extensions or MIME types etc., we can exploit properties of the particular type of data more deeply. For instance, XML documents involve references to respective XML schemas they are supposed to be valid against, XQuery queries refer to the the documents they are posed over, whereas XSDs can refer to other schemas they consist of using constructs such as `import`, `include` or `redefine`. Also more advanced linking XML technologies, such as XPointer [122] or XLink [123] can be used to mutually refer the data.

Even more advanced crawler can deal with typical situations when the referenced data are not present directly in the given address, but "close" to it, i.e., in a neighboring directory, in a file with a slightly modified name etc. In this case the search strategy cannot be exact, but some kind of fuzzy searching and "guessing" must be incorporated. A similar situation occurs in case of, e.g., XQuery queries, where we usually know the exact name of the queried document, but not the path.

Last but not least situation to be solved occurs in situations when a given file type (e.g., a script with XPath queries) does not have a specific extension or a user does not know and use it. So the crawler cannot rely on the extensions and/or other simple types of identification and must analyze directly the content of the file. Naturally, such analysis cannot be detailed since it would highly worsen efficiency of crawling, but a kind of reasonable heuristics for particular file types must be proposed and, in particular, tested.

Improvements of Analytical Plugins The current analytical plugins are able to analyze basic structural aspects of XML documents and XML schemas. Naturally, they can be further extended so that they cover most of the statistics used in the related work (see Section 2.8). However, since the current approaches are relatively old, we can go even further and analyze new, not considered or advanced features. An example can be new constructs of *XML Schema 1.1* [124, 125] such as, e.g., `assert` and `report` that enable one to express advanced integrity constraints using XPath, XSLT scripts and their constructs, complexity and expressive power, or advanced schema languages, such as *Schematron* [51] or *RELAX NG* [102].

Detailed Analysis of Current Real-World Data Having such a robust tool for analysis of real-world data, a natural next step is to perform a detailed analysis of the current real-world data. Considering the XML data, we have focussed on in our first use case implemented in the plugins, we can proceed in several steps. Firstly, a detailed analysis that would cover all the metrics and observations from the existing papers on data analyses (as described in Section 2.8) can be performed, whereas the found differences would bear highly useful and interesting information.

In the second step, and in combination with the previous open issue of advanced crawling, we can focus on analysis of real-world XML operations, in particular XPath and XQuery queries. To our knowledge, there exist no such results, while, on the other hand, the knowledge of typical queries used in the real-world applications would highly help in the respective optimization strategies. An interesting target analysis would be also usage of the queries within other XML technologies, e.g., XPath queries in XSLT scripts or XSDs, XSDs in Web Services [70] etc.

And, last but not least, an important aspect of statistical analyses of real-world data, not only XML one, is analysis of their evolution. A periodical, e.g., monthly, report of results and their aggregation would bear even more important information on evolution and tendencies of XML applications and, hence, could be used for more advanced optimization purposes.

Analysis of Other Kinds of Data Despite the fact that XML data still keep a leading role in data representation and the related XML technologies are robust and mature, there exist other important formats and data types that become more and popular. A classical example are data types related to Semantic Web [126], such as RDF triples [72], ontologies [127], Linked Data [89] etc. In this case we need to solve similar issues, i.e., crawling, correction, and analyses, whereas other aspects, namely evolution are even more important.

2.10 Conclusion

The main aim of this paper was to introduce a complex, open and extensible system called *Analyzer* and describe several related research problems we have focussed on. *Analyzer* allows for performing the full process of data analysis that consists of the following steps:

1. data crawling,
2. data correction,
3. application of analyses, and
4. aggregation and visualization of results.

As a first use case we implemented and tested modules for analyses of real-world XML documents, XML schemas and XQuery queries. In the first three steps of the process we focussed in more detail especially on issues of efficient crawling of XML data, re-validation of invalid XML documents, exploitation of similarity of XML data in data analysis, and XML query analysis.

Despite our original motivations related to XML technologies, we finally implemented an application that is completely capable of performing analyses over documents of whatever types. *Analyzer* represents a framework, that gives a user an environment for gathering documents, configuring analyses, managing and scheduling computations, permanent storage for files and computed data, and a browser for presenting generated reports. The key advantages of *Analyzer* are as follows:

- Multiple versions of the same document are supported,
- Documents can be described by multiple types concurrently,
- Automatic attempts to download referenced documents are performed,
- Projects can be forced to process only documents of selected types,
- All analytical logic is implemented separately in plugins,
- Executing scheduled tasks in multi-threaded environment is exploited,
- Started computations can be interrupted and resumed later, and
- Computed data are permanently stored and available for browsing.

Our future plans will primarily be targeted to issues discussed in Section 2.9. Firstly, 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.

3. Linked Data Analysis

To demonstrate the versatility of Analyzer *we focused on analysis of highly linked data sets from available dumps of the Linked Data Cloud. In this section we describe a statistical analysis of existing real-world RDF triples, in particular the linkage of sample data sets and a structure of triples. We design own metrics for description of structure and depth of linkage to be used in the tuning process of a distributed linked data query engine [61]. The analysis proposes different characteristics that appropriately capture and describe structural features of RDF triples, and provides experimental results over 5 independent data sets with more than 21 millions of RDF triples.*

The contents of this section was published as a conference paper Analyses of RDF Triples in Sample Datasets *[54] at the 3rd International Workshop on Consuming Linked Data (COLD 2012) held in conjunction with the 11th International Semantic Web Conference (ISWC 2012).*

3.1 Introduction

Linked Data [89] is not any particular standard, it is just a set of common practices and general rules using which we can contribute to the Web of Data that emerged recently to enrich the traditional Web of Documents. So, what are these rules? First of all, each real-world entity should be assigned a unique URL identifier; these identifiers should be dereferenceable by HTTP to obtain information about these entities; and, finally, these entity representations should be interlinked together to form a global Linked Data cloud.

Nevertheless, despite there are also other ways how to follow the mentioned Linked Data principles, the most promising is obviously the RDF standard [19]. It assumes data modelled as triples with three components: *subject*, *predicate* and *object*. These triples can also be viewed as graphs, where vertices correspond to subjects and objects, while labelled edges represent the triples themselves.

One of our ongoing research efforts should result into a proposal of a new querying system dealing with large amounts of distributed and dynamic RDF data – issues we previously identified as open problems of the existing approaches from the area of RDF triples storing, indexing and querying [32]. It is apparent that, having the knowledge about structural and other features of data we want to process, we are able to manage such data more efficiently.

In fact, this idea predetermines the aim of this paper – we propose a set of characteristics of RDF triples and provide experimental results over several selected data sets. These characteristics capture features of individual triple components, triples themselves and also structural features of RDF graphs, while performed experiments attempt to outline the nature of real-world RDF data.

3.2 Motivation

If we knew characteristics about data we want to process, we would have better chances to propose algorithms and data structures that could be more efficient with respect to our expectations. In other words, this idea justifies the aim of this paper. Having understood RDF triples we want to store, index and query, we can, hopefully, achieve better results. Moreover, we can also come across approaches that require sort of a configuration (e.g., Structure Index by Tran and Ladwig [37] or Summary Index by Harth et al. [128]). But how can we provide required parameters, if we do not know enough about data or queries?

Therefore, we have proposed several characteristics we find interesting to study. First of all, the majority of indexing approaches (e.g., Hexastore by Weiss et al. [33] or BitMat Index by Atre et al. [34]) proposes to store components of RDF triples and triples themselves separately (even using fairly different structures) in order to reduce space requirements. Knowledge of string features of these component values could support this practice.

The second group of characteristics worth of studying is related to query evaluation and, in particular, access patterns to individual triple components. In case of full-text querying, we usually do not care which particular triple component should match the queried value, but in case of structural querying like SPARQL [35], we need to have suitable indices allowing us to efficiently access particular components according to the prompted query. These indices can be built, for example, on nested lists (Hexastore [33]) or B^+-trees (RDF-3X by Neumann and Weikum [36]).

Finally, we can even attempt to study more complex characteristics based on structure of RDF graphs. When using SPARQL with queries based on graph patterns, we often need to do operations similar to traditional joining in relational databases, only with the difference that we are working with RDF triples, i.e., graph data. This joining can be supported by appropriate indices as well. Like, for example, precomputed paths (RDF-3X [36]) or stars (Structure Index [37]).

It is apparent that this paper cannot encompass all possible features of RDF data that influence possibilities of their processing. So, as we will see in the following section, we have proposed at least several of them (those we treat as the most important ones with respect to our research intent) and attempted to compute them over particular selected real-world data sets.

3.3 Analyses

Having described our motivation, we can move forward to the core part of this paper. First, we provide some essential definitions in order to describe basic knowledge and theoretical background we need to understand to correctly introduce characteristics of RDF triples and data sets we want to study.

3.3.1 Basic Definitions

RDF triples are composed from three components: a subject, a predicate and an object. Beside literal values, the main building block for components of these triples is based on URI (uniform resource identifier) references as they are expected by the RDF standard. However, we assume that these references are always automatically translated to full URIs.

Thus, we can introduce $\mathbb{U}$ as a domain of all possible URI values, i.e., identifiers of resources. Analogously, assume that $\mathbb{B}$ is a domain for blank nodes and $\mathbb{L}$ a domain for literals. We do not need to study the content of these domains, we only use them to restrain the allowed values of individual triple components.

Definition 9 (RDF Triple)**.** *We say that $t = (s, p, o)$ is an* RDF triple *(or just a* triple*), if $s \in \mathbb{U} \cup \mathbb{B}$ is a* subject*, $p \in \mathbb{U}$ is a* predicate*, and $o \in \mathbb{U} \cup \mathbb{B} \cup \mathbb{L}$ is an* object*. We say that t is a* data triple *if $o \in \mathbb{L}$.*

All values (we call them *terms*) from domains $\mathbb{U}$, $\mathbb{B}$ and $\mathbb{L}$ are seen as ordinary strings. This allows us to get deeper insight into the internal structure of URIs, generally conforming to $SchemeName$: $HierarchicalPart$ [? $Query$] [# $Fragment$] scheme (we came across and studied only URLs, thus we could make this simplification). First of all, having any term x, $length(x)$ denotes a *length* of x, i.e., number of symbols it is composed of.

Now, we describe how to split URI terms into two parts. Assume that $x \in \mathbb{U}$ and p is a position of the last # symbol in x. Then we define $prefix(x)$ as a substring of x before p and $suffix(x)$ as a substring after p. If there is no $Fragment$ part, then we analogously use the last occurrence of / symbol from the hierarchical part instead. This approach should capture the way how URI terms are usually used and designed by creators of data documents and ontologies.

Sets of RDF triples are commonly modelled as *RDF graphs*.

Definition 10 (RDF Graph)**.** *Given a set of triples T, we define $\mathcal{G} = (V, T)$ to be an* RDF graph *(or just a* graph*) as follows:*

- *V is a set of graph* vertices*, where $V = \{\, x \mid \exists t \in T,\ t = (s, p, o)$ such that $x = s$ or $x = o \,\}$, and*
- *T as a set of directed graph* edges *corresponds to the underlying set of triples.*

Although we use a term graph, RDF graphs are in fact directed multigraphs since there can be more edges between the same vertices. Next, given a vertex $v \in V$ and an edge $e = (s, p, o) \in T$, we say that e is an *ingoing edge* to v if $v = o$, and that e is an *outgoing edge* from v if $v = s$.

3.3.2 Proposed Characteristics

According to the discussed motivation, we are now able to propose several characteristics that may be useful to know about RDF data we want to store, query or process in a different way.

Term Features

The first group of proposed characteristics is connected with features of individual terms in triples. First of all, the majority of existing approaches for indexing and storing RDF data attempts to find methods of reducing the space required to store the triples. For this aim we can exploit an idea that terms often repeat, or at least their substrings may often repeat across triples in a data set.

In other words, we can inspect lengths of particular terms, either with respect to their type ($\mathbb{U}$, $\mathbb{B}$ and $\mathbb{L}$ domains), or altogether. Next, we can split terms according to our definition of their prefix and suffix parts, exploring one suitable way of finding shared substrings.

Triple Features

Now, we focus on characteristics of triple components and their categorisation. Suppose that we have a set of triples T. Given a particular term x (regardless its type), we may be interested how many triples contain this term at a particular component (subject, predicate or object). In other words, given a suitable term x, we can define $Projection_{s=x}(T) = \{\, t \mid t \in T,\ t = (s,\ p,\ o)$ and $s = x\,\}$ as a *subject projection* (or just *S projection*) corresponding to the set of all triples in T having the given fixed subject value equal to x. Analogously, we can define $Projection_{p=x}(T)$ and $Projection_{o=x}(T)$ as *P projection* and *O projection* respectively. If we model T as a graph, S and P projections correspond to sets of outgoing and ingoing edges respectively.

Moreover, there is no problem extending this idea to projections on two components concurrently. Therefore, we can define *SP projection*, *PO projection* and *SO projection* analogously. For example, $Projection_{s=x,p=y}(T) = \{\, t \mid t \in T, t = (s, p, o), s = x$ and $p = y\,\}$ for two suitable terms x and y. In particular, the SP projection is directly connected with the issue of multivalue properties of RDF triples causing problems in relational databases.

Star Patterns

Let $\mathcal{G} = (V, T)$ be a graph and $v \in V$ a vertex. We define a graph *star* to be a set of edges $\mathcal{S}_v = \mathcal{S}_v^{in} \cup \mathcal{S}_v^{out}$, where $\mathcal{S}_v^{in} = \{\, e \mid e \in T, e = (s, p, o)$ and $v = o\,\}$ is an *ingoing star* around v composed from ingoing edges to v, and, analogously, $\mathcal{S}_v^{out} = \{\, e \mid e \in T$, $e = (s, p, o)$ and $v = s\,\}$ is an *outgoing star* around v.

Next, we define $sig(\mathcal{S}_v)$ as a *signature* of star $\mathcal{S}_v$ (regardless full, ingoing or outgoing) to be a set of all predicates involved in a given star; in other words, $sig(\mathcal{S}_v) = \{\, x \mid t \in \mathcal{S}_v, t = (s, p, o)$ and $x = p\,\}$.

Given a graph $\mathcal{G}$, we can split its vertices V into disjoint sets according to star signatures. This means that two vertices $v_1, v_2 \in V$ belong to the same set, if $sig(\mathcal{S}_{v_1}) = sig(\mathcal{S}_{v_2})$. Since this classification is an equivalency relation over V, we can call these sets as *star classes*. Analogously, we could introduce ingoing/outgoing star classes considering only ingoing/outgoing edges respectively.

Star classes and their sizes can describe uniformity of graph vertices, thus, we can base additional characteristics on the notion of stars. Apparently, their idea is connected (and inspired) by Tran et al. [37] and their Structure Index.

Path Patterns

Let $\mathcal{G} = (V, T)$ be a graph for a set of triples T and $v_S, v_T \in V$ two vertices. We say that a sequence of edges $\mathcal{P}_{v_S,v_T} = \langle e_1, ..., e_n\rangle$ with *length* $n \in \mathbb{N}_0$ is a directed *path* from the *source vertex* v_S to the *target vertex* v_T, if the following conditions hold:

- First, let $\forall\, k \in \mathbb{N}, 1 \leq k \leq n$: $e_k = (s^k, p^k, o^k)$ and $e_k \in T$.
- If $n > 0$, then $s^1 = v_S$ and $o^n = v_T$. If $n = 0$, then necessarily $v_S = v_T$.
- Next, $\forall\, k \in \mathbb{N}, 1 \leq k < n$: $o^k = s^{k+1}$, i.e., edges follow each other.
- $\neg\,\exists\, j, k \in \mathbb{N}, 1 \leq j < k \leq n$: $s^j = s^k$ or $o^j = o^k$ or $s^j = o^j$, in other words, vertices do not repeat.

Given a particular path $\mathcal{P}_{v_S,v_T}$, we can define its *signature* as a sequence of predicates of its edges, i.e., $sig(\mathcal{P}_{v_S,v_T}) = \langle p^1, ..., p^n \rangle$.

Directed paths can serve as another characteristic that is closely related to the process of evaluating queries based on SPARQL graphs patterns.

Features Summary

The following listing provides a simplified overview of all characteristics over RDF triples we have proposed in this paper:

- Term lengths – length of $\mathbb{U}$ and $\mathbb{L}$ terms viewed as strings.
- Term prefixes – length of prefixes and suffixes of $\mathbb{U}$ terms.
- Data triples – ratio of data and other triples in data sets.
- Triple projections – cardinality of S, P, O and SP, PO, SO projections.
- Star patterns – sizes of graph, ingoing and outgoing star classes.
- Path patterns – path occurrences according to their signatures.

3.4 Experiments

In this section, we first describe publicly available data sets we have chosen for our experiments, then we provide their implementation basics and, finally, we present results over these data sets together with some general observations.

3.4.1 Data Sets Selection

The selection of appropriate data sets is probably one of the most important issues of any experiments. The first option could be to download a representative sample of RDF triples from the entire Linked Data cloud. However, with respect to the planned usage of our querying framework, we have finally decided to perform the experiments over a few selected data sets only. They are from different sources, cover different thematic areas and they contain several millions of triples. Although we cannot omit *DBPedia* as one of the most important Linked Data sources, we selected also other interesting ones. In particular, data sets that are listed in the following summary, including their abbreviations we will use in the further text:

- ACM (ACM publications[1]) – ACM proceedings data set with author and publication information.
- DBCS (Czech DBPedia[2]) – information extracted from Czech Wikipedia infoboxes. This data set contains less clean data, which is actually a common situation in sources that are automatically derived from non-structured data.

[1]`http://acm.rkbexplorer.com/models/acm-proceedings.rdf`

[2]`http://downloads.dbpedia.org/3.7-i18n/cs/infobox_properties_cs.nt.bz2`

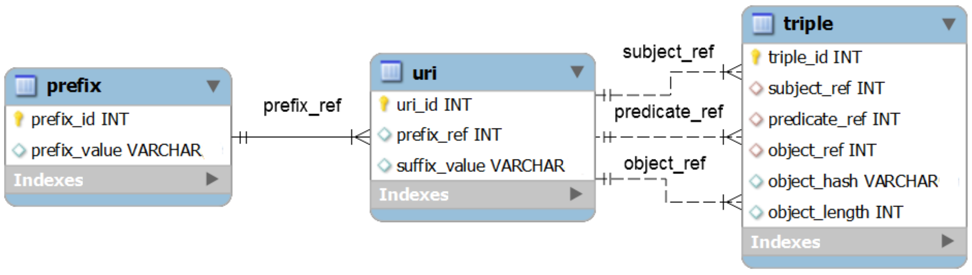


Figure 3.1: Database schema

- DBEN (English DBPedia[3]) – information about persons (records like date and place of birth etc.) extracted from English and German Wikipedia, represented using the FOAF vocabulary.
- GO (Gene Ontology[4]) – one of the data sets of Bio2RDF project describing publicly available DNA sequences.
- MDB (Movie Database[5]) – database containing triples about actors, movies and their relationships.

3.4.2 Implementation Basics

We downloaded dumps of all the previously described data set in one of these formats: RDF/XML[6], n-triples[7] or Notation 3[8]. Then we parsed these dumps using scripts[9] implemented in Java and Python.

After necessary data cleaning (some data sets contained syntax errors), we stored all obtained triples into MySQL database using Percona Server 5.5[10] running on Debian operating system.

Since we wanted to achieve efficient computation of the proposed characteristics, we designed the database schema so as to be based on three tables: the first table contains all URI prefixes, the second one full URI values, and, finally, the third one contains triples themselves. However, instead of URI terms we stored references to the second table and instead of literals it contains their MD5 hashed values together with original lengths. The simplified schema is shown in Figure 3.1.

3.4.3 Experiment Results

The majority of proposed characteristics was computed using MySQL scripts[11]. The description of the most interesting observations together with detailed experiment results is the subject of the following text.

[3]`http://downloads.dbpedia.org/3.7/en/persondata_en.nt.bz2`
[4]`http://s4.semanticscience.org/bio2rdf_download/rdf/genbank`
[5]`http://queens.db.toronto.edu/~oktie/linkedmdb/linkedmdb-latest-dump.nt`
[6]`http://www.w3.org/TR/rdf-syntax-grammar/`
[7]`http://www.w3.org/TR/rdf-testcases/`
[8]`http://www.w3.org/TeamSubmission/n3/`
[9]`http://ksi.mff.cuni.cz/~starka/ld_parsers.zip`
[10]`http://www.percona.com/software/percona-server/`
[11]`http://ksi.mff.cuni.cz/~starka/ld_mysql.zip`

General Characteristics

Firstly, we present the basic characteristics of the data, the number of unique prefixes, URIs and triples. These results show the diversity of triples within particular data sets (see Table 3.1).

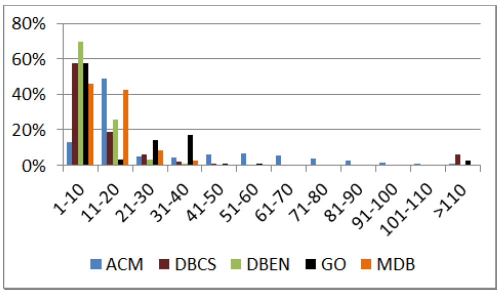
Table 3.1: Term and triple characteristics

	ACM	DBCS	DBEN	GO	MDB	Total
Term Counts						
Unique prefixes	11	10,157	137	195	5,204	15,704
Unique URIs	810,266	162,625	867,428	1,187,775	1,327,165	4,355,259
Triple Counts						
Unique triples	2,715,890	1,426,244	4,502,983	7,411,868	5,291,548	21,348,533
Data triples	840,008	1,019,355	3,006,569	2,418,975	2,418,413	9,703,320
Term Lengths						
Term prefixes	31.55	54.52	48.87	57.27	47.66	52.21
Term suffixes	30.07	18.12	16.87	19.03	16.23	19.77

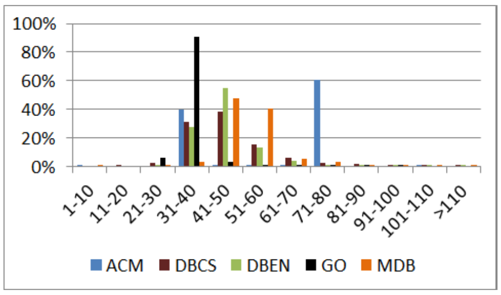
We can see that there are only 11 unique prefixes in ACM data set, whereas there are 10,157 prefixes in DBCS data set. These numbers suggest that ACM data set is relatively closed (it contains mainly entities within its own domain – publications and their authors), while DBCS contains many *dirty* triples, i.e., triples where the object component is recognized as a URI but it is not a part of *DBPedia* (and probably neither a part of the Linked Data cloud).

The results also show the average lengths of URIs. Although there are no extreme values, we can see that ACM data set differs. This is because the URIs (in all data sets) often contain artificial and often automatically generated identifiers combined with entity types and/or human readable names.

The detailed distribution of both URI and literal term lengths with respect to selected data sets can be seen in Figure 3.2. Since each data set has a different total number of triples, we normalized the computed lengths by the total number of terms in each data set.



(a) Literals



(b) URIs

Figure 3.2: Distribution of literal and URI term lengths

As we can see, all data sets except ACM use URIs around 40 characters long. This is because identifiers in ACM data set are padded by numeric values which causes these URIs are of the same length. On the other hand, the lengths of literals mostly range from 1 to 20 characters. This is caused by the usage of common values, i.e., person names, dates, numbers, etc. Only ACM data set contains textual literals, in particular, keywords and concatenated lists of authors.

Triple Projections

Assume that T represents a particular data set, Θ is a comparison operator over $\mathbb{N}$ (e.g., $=$ or $>$) and $c \in \{s, p, o\}$ stands for particular triple component. Then we can define $size_{\Theta z,c} = |\{\, x : |Projection_{c=x}| \;\Theta\; z\,\}|$ as a shortcut for the number of terms x whose projections $Projection_{c=x}$ according to a particular component c have exactly z triples in case of $=$, more than z triples in case of $>$ (and analogously for the other comparison operators).

We can also define $size_{\Theta z,c_1,c_2} = |\{\,(x_1,\, x_2) : |Projection_{c_1=x_1,c_2=x_2}| \;\Theta\; z\,\}|$ for double projections with both c_1, $c_2 \in \{s, p, o\}$ and $c_1 \neq c_2$ as expected.

These two notions help us to present interesting features of the triple projection characteristics. In other words, we study the distribution of terms (or pairs of terms in case of double projections) according to their significance inside given data sets. For example, having the condition Θz equal to $= 1$ and inspecting the object components, we are interested in terms x such that there exists right one triple in T with x at component o. Then, $size_{=1,o}$ gives us the number of such x in T. Several projection results are presented in Table 3.2.

Table 3.2: Triple projections

	ACM	DBCS	DBEN	GO	MDB
Term Counts					
Unique subjects	810,248	68,946	790,703	776,698	694,399
Unique predicates	14	5,227	9	22	250
Unique objects	489,912	98,638	76,736	1,171,491	1,049,248
Simple Projections					
$size_{=1,o}$	403,425 (82.3%)	69,886 (70.9%)	45,551 (59.4%)	888,005 (75.8%)	657,484 (62.7%)
$size_{>1,o}$	86,487 (17.7%)	28,752 (29.1%)	31,185 (40.6%)	283,486 (24,2%)	391,764 (37.3%)
Double Projections					
$size_{=1,s,p}$	2,002,042 (89.1%)	881,317 (86.4%)	246,078 (94.2%)	6,429,816 (98.8%)	4,386,514 (94.5%)
$size_{>1,s,p}$	245,828 (10.9%)	139,254 (13.6%)	3,964,721 (5.8%)	79,925 (1.2%)	253,471 (5.5%)
$size_{=1,p,o}$	403,425 (82.3%)	102,127 (79.2%)	56,018 (61.7%)	910,420 (76.2%)	873,643 (74.1%)
$size_{>1,p,o}$	86,487 (17.7%)	26,761 (30.8%)	34,809 (38.3%)	284,764 (23.8%)	306,150 (25.9%)
$size_{=1,s,o}$	2,661,787 (99.1%)	381,569 (82.6%)	1,439,886 (64.0%)	4,980,199 (86.4%)	2,857,318 (80.9%)
$size_{>1,s,o}$	24,297 (0.9%)	80,359 (17.4%)	810,836 (36.0%)	783,039 (13.6%)	673,347 (19.1%)

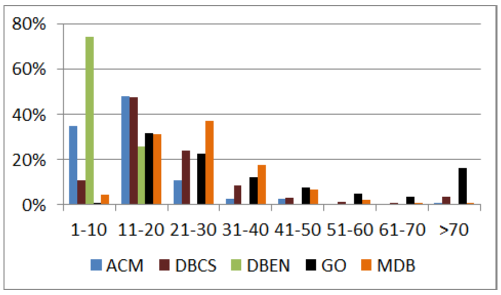
The results show, that there are usually only few unique predicates which are used in the triples. In DBCS, there are over 270 triples for each predicate, which is the lowest ratio between all data sets. In other data sets, there are thousands of triples per predicate. For subjects, the average number varies from 2 to 20.

In the second and third part of the table, we show projections for O and SP, PO, SO respectively. In each case we split the entire space into two disjoint parts: classes with size equal to 1 and classes with greater size. It is interesting that in most cases the projections usually have right one triple. We can also say that a typical data set contains only a very limited number of predicates. Subjects are used mostly more than once, but they do not form large hubs.

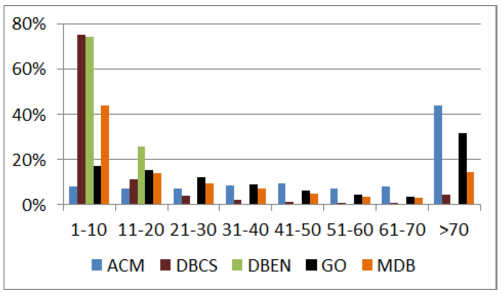
Star Patterns

Assume that T are triples of a particular data set, then we can split vertices V of the corresponding graph $\mathcal{G} = (V, T)$ into star classes according to signatures of their star patterns, as we already know. Figure 3.3 depicts the distribution of star classes according to their sizes, separately for ingoing and outgoing stars. In other words, e.g., for ingoing star patterns, the horizontal axis represents different possible sizes of signatures (different numbers of predicates on ingoing edges) and the vertical axis represents the overall number of ingoing star classes having the given size.

The values are normalised in the same way as in the term length characteristic, i.e., normalised by the total number of distinct star signatures in the particular data set.



(a) Outgoing



(b) Ingoing

Figure 3.3: Distribution of ingoing and outgoing star class sizes

We can see that most of the unique outgoing stars have the size (i.e., number of outgoing predicates) from 10 to 30. Similarly, most of the ingoing stars have the size from 10 to 30, only except ACM data set where sizes are distributed uniformly. Moreover, for all data sets except DBCS, the first 10% of star signatures covers more than 80% of triples.

Path Patterns

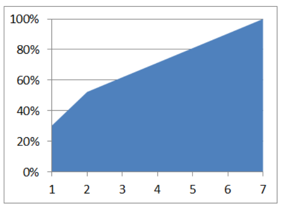
Similarly to star patterns, we computed also the path pattern characteristics. In particular, we considered paths of lengths equal to 2 and 3, since longer paths were out of our computation possibilities. For each path length we detected the number of unique path signatures and the overall number of all paths conforming to them, as we can see in Table 3.3.

Moreover, we also studied another aspect – having a particular number of the most frequent path signatures, how many paths do these signatures conform to? The number of paths with the most frequent signature is presented in the mentioned table, while the entire dependency is depicted in Figure 3.4.

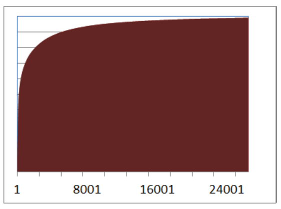
Table 3.3: Statistics of path classes

Length	Property	ACM	DBCS	DBEN	GO	MDB
2	Unique signatures	7	33,394	14	55	275
	Number of paths	3,382,538	1,191,731	178	1,300,120	2,470,993
	Greatest class	1,026,874	27,786	38	247,477	248,633
3	Unique signatures	0	67,107	0	206	664
	Number of paths	0	1,428,871	0	26,863,416	15,804,941
	Greatest class	0	15,531	0	2,754,908	550,887

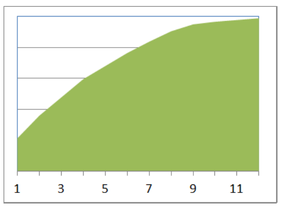
Finally, according to computed results, the ratio between unique path signatures and all paths themselves is relatively low. In other words, having a particular frequent signature, there are many paths conforming to it, which can be exploited in indexing techniques dealing with precomputed paths.



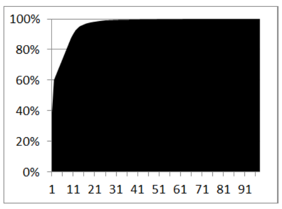
(a) ACM



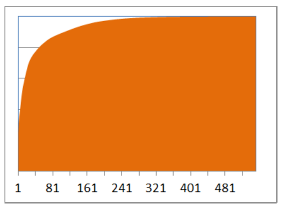
(b) DBCS



(c) DBEN



(d) GO



(e) MDB

Figure 3.4: Aggregated number of paths according to the signature frequency

3.5 Related Work

Although there exist several works about analyses of semantic documents and Linked Data, there are still open questions that could be discussed.

We start this overview with a system proposed in Section 2 where we proposed a system for automatic document acquisition and analysis. Although we primarily focused on structural characteristics of XML documents, some basic ideas and insight into the complexity of exported data sets can be applied also in the context of Linked Data.

Ding et al. [38] described the analysis of more than 1.5 million FOAF documents. In particular, they inspected the usage of the FOAF namespace, host names and particular properties, as well as the relationships of a person in a group and other components of a social network. In general, this work describes several interesting characteristics, but its impact and context is very restrained.

Both the previous works assumed analyses at the document level, whereas Rodriguez [129] looked at data sets from the Linked Data cloud in a more complex way and computed some basic characteristics between them.

The general statistics of the Linked Data cloud are described in Bizer et al. [39]. The authors aimed at characteristics and link statistics between selected data sets. These data sets were divided by different thematic domains, for which several ingoing and outgoing statistics were computed. Provenance, licensing and data set-level metadata published together with these data sets were also considered.

3.6 Conclusion

In this paper we focused on several characteristics of publicly available Linked Data data sets. The results show that although the data sets are from different areas, published by different methods and institutions, some of their characteristics are similar and, thus, the knowledge of these characteristics can be harnessed to make the management of RDF data more efficient.

We considered only a small sample of the Linked Data cloud as well as only a limited set of proposed characteristics dealing primarily with RDF triple components and structure only. On the other hand, we hope that despite this fact some observations presented in this paper can be generalised, further extended and appropriately exploited. In our future work, we plan to enrich these characteristics and also encompass a wider set of data sets and triples themselves.

4. XML Query Analyses

As we have mentioned, there exist several papers dealing with analyses of XML data. However, currently there exists no paper that focuses on XML data operations, namely XML queries. From the perspective of the implementor of an XQuery or XSLT processor, such an analysis may discover that a number of language features is rarely used and, therefore, not worthy of aggressive optimization. For example, the following/preceding *axes are supposed to be used significantly less frequently than the* child/descendant *axes; consequently, the majority of indexing and querying techniques like twig joins [29] are limited to the* child/descendant *axes.*

In this section, we describe the design and the implementation of a module for statistical analyses of XQuery programs, as an extension of our previously described analytic framework Analyzer *(see Section 2). In particular, the XQuery program is converted to an XML representation which allows the user to formulate analytical queries in XPath. Preliminary experimental results computed over the* XML Query Use Cases *[30] and* XML Query Test Suite *are described as well.*

The contents of this section was published as a conference paper XQConverter: A System for XML Query Analysis *[58] at the 6th International Workshop on Flexible Database and Information System Technology (FlexDBIST 2011) held in conjunction with the 22nd International Conference on Database and Expert Systems and Application (DEXA 2011).*

4.1 Introduction

Since the number of implementations of XQuery [21] evaluators is significantly growing, the implementers encounter problems of optimization of different XQuery constructs. The question is which of them are worth to engage, i.e., which are used in real-world queries often and which are not. In this paper we describe a system that will answer the question. The tool is able to count the number of selected constructs and their combinations over a given XQuery program and, hence, to provide information on typical XQuery queries.

A typical approach of researchers analyzing programming languages is to implement a simple tool for recognizing words in the target language, summarize these words and so get the frequency of their usage. Due to extreme *context dependency* of XQuery, such simple tools cannot work correctly and lead to wrong results. For example, a short and syntactically correct XQuery program

```
for $for as for in "for" return <for/>
```

contains the word "`for`" five times, each time in a different meaning depending on its position in the program. The researcher with a simple tool that is probably looking for the word "`for`" in the meaning of a *for*-clause from the *FLWOR* expression gets misleading results. That is why a more precise tool needs to be implemented, at least as strong as a correct *lexical scanner* of XQuery, which can also recognize the meaning of the scanned words. However, such scanner would be sufficient for calculating usage frequency of individual words with the given meaning, but not whole language constructs and their combinations, such as, e.g., the number of multiplicative operators within FLWOR expression. Hence, we also need an *XQuery parser* connected to the lexical scanner which would be able to recognize the structure of programs according to XQuery grammar. The selected constructs can be then traced by adding a simple code for accumulating the number of passes over the given rule representing the selected constructs in the parser. Nevertheless, the disadvantage of this solution is that tracings of constructs are hard coded and when it is required to add or change the observed construct, the source code of the parser needs to be modified. This conclusion leads us to an idea of *general observation*, i.e., creating a data structure representing the input program together with a meta-language for querying the structure.

A simple and natural way we utilize in this paper which enables one to reach this target is to build a *syntax tree*. It can be then easily transformed into XML [1] representation and queried using, e.g., XPath [28]. However, this approach expects the analyzer to know the full and complex XQuery grammar. In addition, a syntax tree is too big even for small XQuery programs and contains parts that are not interesting for the analysis. That is why we propose a new and more suitable representation of an XQuery program that suppresses the unimportant aspects and simplifies further processing. In general, we describe the design and implementation of a general system for the analysis of XQuery programs, *XQConverter*, as the extension of our previously described analytic framework *Analyzer* [60] which, among others, offers friendly user interface for creating new queries and browsing the results.

Related Work. Although the analysis of the XML data and its related formats is one of the common optimization strategy (such as [14, 2]), the complex analysis of real XQuery programs is still missing. Currently, there exist a test suite[31] and several benchmarks (a short survey described in [130]) allowing to test XQuery engines, but they do not respond to the question of the used constructs and their distribution in domain specific sets. In this paper, we propose the extension of the *Analyzer* framework to represent a XQuery program which allows the user to make statistical queries.

4.2 XML Representation

In the following sections there are described the basics for designing the internal XML representation according to which it is decided how the given construct should be formed in the XML representation and, hence, what nodes and attributes will be used. Not all constructs of XQuery are described in detail, but only those which are complex and interesting and where different possible approaches can be discussed. The presented constructs are introduced in a natural order. For instance an expressions needs to be designed before operators, because the operators use the expressions. Due to the size restrictions the full XQuery grammar, referenced in following text, is defined in the *XQuery grammar*[1].

4.2.1 Types

Types in XQuery are represented by nonterminals *SequenceType* and *SingleType* introduced in rules 117 and 119 in the XQuery grammar. Syntactically and also semantically, *SingleType* is also *SequenceType* as it consists of *AtomicType* and an optional

[1]`http://www.w3.org/TR/xquery/#nt-bnf`

```
document-node(element(ns:root,ns:RootNodeType?))

<Type cardinality="one">
  <KindTest kind="document">
    <KindTest kind="element" name="ns:root"
        type="ns:RootNodeType" nillable="true"/>
  </KindTest>
</Type>
```

Figure 4.1: Examples of types

occurrence indicator. This allows unification of constructs for these nonterminals into a single node called *Type*. For distinguishing these two types in the XML, it has to be mentioned that *SingleType* appears only in productions of symbols *CastableExpr* and *CastExpr* in rules 56 and 57. With the exception of an empty sequence, the sequence type consists of *ItemType* and an optional cardinality, which express the number of items in the sequence. When the cardinality is not specified explicitly, it is of one by default. The fact that the type is an empty sequence can be also expressed as it has cardinality of zero. So the first information captured in *Type* node is the attribute called *cardinality* and its value can be one of `zero` (for the empty sequence type), `one` (if the cardinality is not explicitly specified), `zero-or-one` (for the symbol `"?"`), `zero-or-more` (`"*"`) and `one-or-more` (`"+"`).

As rule 121 exposes, the type of items in the sequence can be either any item type (terminal `"item()"`), an atomic type (such as `xs:integer`), or a node type (nonterminal *KindTest* in rule 123). The first two can be easily converted into individual nodes *AnyItem* and *AtomicType*, where *AtomicType* node holds one attribute *name* containing the name of the built-in atomic type. The last rule that needs to be converted into an XML structure to cover the whole construction of types in rule 123 with all types of nodes. They can be expressed via a single node called *KindTest* with a specified type in a value of attribute *kind*, i.e., *document*, *element*, *attribute*, *schema-element*, *schema-attribute*, *processing-instruction*, *comment*, *text* or *any-kind*. Some of these node types bear additional information, such as the name of an element or an attribute and its data type, which are not further structured, so they can be expressed also as attributes *name* and *type*.

The Figure 4.1 shows an example how the type is converted into its XML representation.

4.2.2 Operators

Operators in XQuery are exposed in production rules from 46 to 58. The order how the rules are nested describes the operator precedence. In our approach, we express the precedence in the representation itself, not in the nodes of operators, with the aid of a *LALR*[2] *parser* [131]. As the parser shifts over the particular rules in the order of operator precedence, the XML representation is built. The preceding operations appear at the lower levels of syntax tree and at the lower levels of XML representation as well. This fact allows us to use unified node *Operator* for all operators with the

[2]Look-ahead LR parser is a parser driven by a parse table in a finite state machine format

$E1$ + $E2$ and $E5$ <= $E6$ to $E7$

```
<Operator class="logical" name="and">
  <Operator class="additive" name="plus">
    nodes for $E1$ and $E2$
  </Operator>
  <Operator class="comparison"
    name="less-than-equals" subclass="general">
      node for $E5$
    <Operator class="range" name="to">
      nodes for $E6$ and $E7$
</Operator></Operator></Operator>
```

Figure 4.2: Operators example

information about operator in attribute *name*. This requires unique values for each operator, but there are operators that can be captured with the same value, as this fact might be interesting for the analysis. For example operator *plus* actually acts as two operators, unary or binary. Also comparison expression operators `"eq"` and `"="` both express equality, but the first one in a value collation and the second one in a general comparison meaning. This leads to an idea of unification of some operators with either same notation or with the similar meaning, leaving the distinguishing information into attribute *class* (i.e., unary, binary, etc.). Optionally, when this is not sufficient for separating the meaning (only in case of comparison operators), the remaining information for distinguishing operators is stored in attribute *subclass*. Usually one (in case of unary operators) or two (binary operators) nodes with *Expr* meaning can appear under *Operator* node.

The example at the Figure 4.2 shows how a more complex expression with multiple operators with the different precedence is transformed into the XML representation.

4.2.3 FLWOR Expressions

The *FLWOR expression*, as the workhorse of XQuery, also deserves a detailed description. Production rule 33 presents FLWOR as a sequence of `for` and `let` clauses (together called a *tuple stream*), an optional `where` clause, an optional `order by` clause and a mandatory `return` clause. Hence, the designed *FLWOR* node will consist of *TupleStream* node, optional *WhereClause* and *OrderByClause* nodes, and *ReturnClause* node.

An interesting situation is in case of *OrderByClause* node. As rule 39 describes, the `order by` clause consists of one or more order specifications. Each order specification consists of *Expr* node and optionally of some modifiers introduced in rule 41. All of these modifiers are scalar, so it is preferred to mention them in the form of attributes. The construction of the order clause can be seen on Figure 4.3.

```
stable order by
  $book/quality collation "http://www.ex.org/q",

<OrderByClause stable="true">
  <OrderSpec collation-uri="http://www.ex.org/q">
    node for $book/quality
  </OrderSpec>
</OrderByClause>
```

Figure 4.3: Example of order by clause

4.2.4 Path Expressions

A *path expression*, captured in rules 68 and 69, consists of one or more *steps*, optionally beginning with `"/"` or `"//"`. The slashes denote the initial step of the path. When it is not specified, the path begins with the *context node*, otherwise the *root node* is the initial step. This aspect can be easily captured as *Path* node with a single attribute called *initial-step* and its values of *root* or *context*.

According to rules 70, 71 and 81, the individual steps are either *axis* or *filter* steps and both of them can be specified with predicates. As rules 82 and 83 show, the predicates are just lists of expressions. This allows for designing *Predicates* node in a very similar way as *CommaOperator* node and, thus, under this node a list of nodes with expression meaning can appear. The step itself, either the filter step or the axis step, is designed as *Step* node and the filter/axis node with optional *Predicates* node is simply appended.

On the other hand, the axis step is a little bit complicated, because of abbreviated syntax. It is expected to represent the abbreviated and unabbreviated axis step as similarly as possible. When the axis with abbreviated form is used, it is converted into its unabbreviated form and then transcribed to the XML representation. All axes then contain boolean attribute called *abbreviated* for expressing the used syntax. Every axis is then transcribed into the XML representation as *Axis* node with attribute *kind* capturing the kind of axis (child, self, etc.) and an additional attribute for axis direction called *direction* that can have two possible values: *forward* and *reverse*. Example of short path expression `//car[@type="SUV"]` and its corresponding XML representation is depicted in the Figure 4.4.

4.2.5 Constructors

The last constructs whose design we describe in detail are *constructors*. A very similar approach, as in Section 4.2.2, is used to create a uniform node, called *Constructor*, for all constructors. There are two construct types: *direct* and *computed*. Each of them can construct several node types. This information can be easily captured in attributes *kind* and *type*. The child nodes of the Constructor node are: *Name* (for the computed and the direct element constructor and the computed attribute constructor), *AttrList* (only for the direct element constructor), *PITarget* (for the direct and the computed processing instruction constructor) and finally *Content* (for all constructors). Thus, each constructor representation consists of some of these four nodes. *Name* node is at the place where either a direct name is presented or is built by an expression. *AttrList*

```
<Path initial-step="root">
  <Step><Axis abbreviated="true" direction="forward"
     kind="descendant-or-self">
      <KindTest kind="any-kind"/>
    </Axis></Step>
  <Step>
    <Axis abbreviated="true" direction="forward"
        kind="child">
      <NameTest name="car"/>
    </Axis><Predicates>
      <Operator class="comparison" name="equals"
       subclass="general">
        <Path initial-step="context">
          <Step><Axis abbreviated="true"
              direction="forward" kind="attribute">
              <NameTest name="type"/>
            </Axis></Step></Path>
        <Literal type="string" value="SUV"/>
      </Operator></Predicates>
  </Step>
</Path>
```

Figure 4.4: Path expression examples

node simply consists of the list of *Attribute* nodes, as rule 97 expresses.

The most interesting part of constructors is *Content* node, as it can appear in all possible constructor types.

The example of the element constructor with a mixed content and its XML representation is shown in the Figure 4.5.

4.2.6 Querying

As we previously mention, we use XPath as an instrument to query the output XML. For instance, the XPath expression depicted bellow evaluates the additive operators in a given XQuery program.

```
//Operator[@class="additive"]
```

4.3 Architecture and Implementation

The architecture of *XQConverter*[3] consists of several parts as depicted in Figure 4.6. It consumes an XQuery program in the form of a character stream. This stream is transformed into lexical tokens by the *XQuery lexical scanner*. The tokens form the input for the *XQuery parser* which instead of building a syntax tree builds the XML

[3]http://kenai.com/projects/analyzer/downloads/download/xqa.zip

```
<tv>five&gt;<program/>{attribute channel {10}}</tv>

<Constructor kind="direct" type="element">
  <Name name="tv"/>
  <Content>
    <String value="five"/><EntityRef name="gt"/>
    <Constructor kind="direct" type="element">
      <Name name="program"/>
    </Constructor>
    <Constructor kind="computed" type="attribute">
      <Name name="channel"/>
      <Content>
        <Literal type="integer" value="10"/>
      </Content>
    </Constructor>
  </Content>
</Constructor>
```

Figure 4.5: Mixed element content

representation of the given XQuery program. Before this XML document is declared as an output, it is *validated* against the respective XML schema[4].

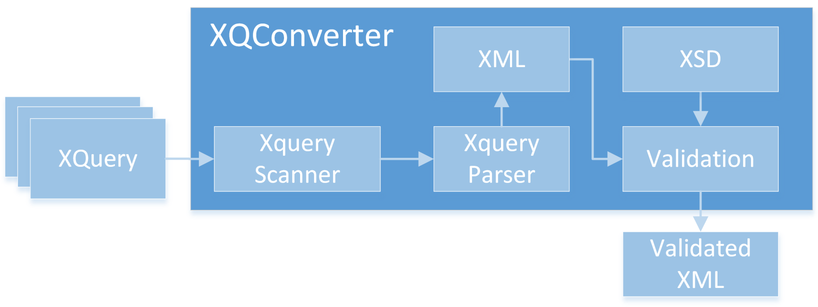


Figure 4.6: Architecture of *XQConverter*

The implementation of the lexical analysis of XQuery is based on technical report [132]. The scanner was implemented using *JFlex* [133], a lexical scanner generator for Java [134]. The XQuery parser was created using *CUP* (Constructor of Useful Parsers) [135], a LALR parser generator, and the Java modification of *Berkeley Yacc BYacc/J*[5].

4.4 Proof of the Concept and Correctness

To prove the concept and correctness of *XQConverter* we tested it using the W3C *XML Query Test Suite* [31], version XQTS 1.0.2. The test suite consists of 15,121 test cases, each testing a single XQuery program. There are four scenarios described in the test

[4]`http://kenai.com/projects/analyzer/downloads/download/xqa.xsd`
[5]`http://byaccj.sourceforge.net/`

catalog: *standard*, *parse error*, *runtime error* and *trivial*. The *standard* and *trivial* scenarios expect the implementation to produce valid results. The *parse error* scenario expects raising a parsing/syntax error at the query parse time. Finally, the *runtime error* scenario expects raising a runtime error at the query run time.

Since *XQConverter* does not evaluate the XQuery program, the *runtime error* scenario is considered as the *standard* scenario which produces valid results. This approach divides test cases into two parts: (1) syntactically correct and (2) raising a parse error. On the other hand, *XQConverter* considers the test case as correct, when it successfully builds the XML representation of the given XQuery program. When *XQConverter* raises the *XQParseException*, the result of the test case is considered as parse error. A single test case is considered as passed when the result (correct or error) matches the expected test result, failed otherwise.

XQConverter does not guard the balance of node names, so the test cases verifying this aspect do not take account. It also strips comments from the input character stream before reaching the lexical scan to avoid mentioned problems with long tokens. Thus, the programs testing forbidden comments in several constructs (such as in the direct constructor content) are not considered as well. All the other aspects, such as the extra grammatical constraints from [21] in Section A.1.2, are tested as usual.

The overall results of the test are listed in Table 4.1. The numbers shows that 99.96% test cases passed, thus we can declare *XQConverter* as correct for analyzing XQuery programs.

Status	**Number**	**Percentage**
Total passed	15,115	99.96%
Total failed	6	0.04%

Table 4.1: W3C XML Query Test Suite results

Preliminary experimental results computed over the *XML Query Use Cases* [30] and *XML Query Test Suite* are described in [62]. A throughout analysis of a representative set of real-world queries is a subject of our current research.

4.5 Conclusion

In this paper we described *XQAnalyzer* – a unique tool for analysis of collections of XQuery programs. It works with a set of XQuery programs and translates them into an intermediate XML code. Subsequently, analytical queries in XPath may be placed over the translations to study the presence, quantity and context of the specific constructs. Using a standard test suite we provided a proof of the concept and its correctness.

In our future plans we will focus on further improvements of *XQAnalyzer* and its exploitation in throughout analysis of both current state of real-world XQuery queries and their evolution in the following months. This intent primarily requires a specific crawler which would enable one to discover the queries efficiently. We also 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.

5. Data Extraction

As described in Section 2, Analyzer *focuses on documents which are imported by a user or automatically downloaded from the Web. Although the download module detects links in HTML or XML documents, it is not able to fulfill the requirement R2 stated in the Introduction, i.e., extraction of data (e.g., URL of relevant pages, or related data stored in another document). Despite it would be possible to implement an improved module for* Analyzer *with the ability to extract additional data from the Web and store them in a database, it does not correspond to the task* Analyzer *was originally created for. Thus we propose a new extraction tool called* Strigil.

Strigil *is a framework for automated data extraction. It represents an easily configurable tool that enables one to retrieve a data from textual or weak structured documents. This section contains description of the framework architecture and its important components. Additionally, we propose a scraping language inspired by the XSLT transformations designed to extract data from different kinds of documents. Despite there are many different approaches focused on various aspects of data scraping, they are usually very specialized to a concrete domain or a data source. We compare these solutions and describe their advantages and disadvantages. Our scraping language is designed to work with an ontology to map scraped data directly to classes and attributes.*

The content of this section corresponds to a conference paper Strigil: A Framework for Data Extraction *[65] that was submitted to the 12th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE 2013), and it is currently under a review process.*

5.1 Introduction

Currently, the World Wide Web is used as a primary source of information. Data are stored in different kinds of databases and presented to the user through various static or dynamic documents, usually queried by HTML [18] forms. The documents are in different formats, e.g., XML [1], HTML [18], PDF, [95] etc. Some of these formats keep structural information of the original database, others mix data in different ways to present them in a more user friendly manner. Although the data are publicly available via a Web interface, they are not available directly to make custom queries, e.g., to get new views over the domain or to find new links between independent data sets. To enable such a functionality, we can simply crawl the data; however, they can be invalid or corrupted, so a correction or an additional post-processing may be required, which leads to the demand for automatic extraction tools that are able to extract and process the resulting data sets automatically (also denoted as *screen scraping*).

For example, we want to make an analysis of public procurements in the Czech Republic, but the government publishes the data about public contracts with a price higher than a limit provided by the law. However, contracts with a lower price can be found at the Web sites of the cities, regions, etc. Despite the good intention of these local Web sites, this information is lost in the number of distinct publication systems, and, for example, it is not possible to ask which supplier won the most public contracts, or which region has the most overpaid procurements, etc. Thus a union of these independent databases is required.

In this paper, we discuss the problems stated in Section 1.1.2, i.e., data extraction from HTML documents, and we propose a way how to integrate data from different sources, and return them as a unified set of RDF triples. We propose a system for data extraction, called *Strigil*, which is driven by our proposed *scraping script language*. We focus on a supervised solution (i.e., driven by a user-defined scraping script) which aims at data fields in any semi-structured format of a particular domain. Although we describe data extraction from HTML documents, the script language is designed to be able to extract data from various document types. The domain is represented by an ontology, e.g., the *Public Contracts Ontology*[1] (PCO), which is used to map the extracted data to its classes and attributes in order to capture their semantics.

Contributions The main contributions of our framework are the following:

- Our prototype system can extract semi-structured data from any HTML document. Although we mention only this format, the system is designed to work with any type of document (e.g., MS Excel or text documents). Since the system is extensible, new *selectors* for other formats can easily be added.
- We support mapping to ontologies describing a domain or any combination of different domains. It allows the user to work with specified data types without the necessity of their explicit declaration in the script.
- The system is self-supervising. Any changes in the document that would cause usage of invalid selectors is detected by checking the ontology coverage.
- The prototype implementation supports basic distributed crawling, including multiple independent downloaders, on multiple computers, working with multiple proxy servers.

5.2 Problem Statement

Currently there are several typical problems relevant to speed, efficiency and universality of the scraping process.

Performance The first group of these problems is related to the performance, i.e., the speed of the connection and the speed of the computer. Obviously the faster the connection is, the more documents are processed. In the context of extraction of data from HTML documents on the Web the parallelism can highly improve the speed of downloading.

Accessibility On the other hand, if the scraper is too aggressive, i.e., downloads multiple documents in parallel and the server is not able to respond to all download requests, the server can block the scraper. To evade this blockage, reasonable *politeness policies* are required. These policies are a set of rules which assure polite crawling of servers, e.g., the policy limits download requests per second. In addition some of

[1]The Public Contracts Ontology enables one to express structured data about public contracts in RDF, in a machine-readable format (`https://code.google.com/p/public-contracts-ontology/`)

the scrapers provide a human interface emulation, which is based on random intervals between download requests (which reduces the load of the server).

Deep Web Additionally, there are problems related to the structure of the Web. One of the most problematic (but well examined) part of the Web is called the *Deep Web*. It is a part of the Web hidden behind HTML forms, JavaScript or different AJAX[2] calls. Extraction systems do not know the context of the page, so they are not able to send an HTML form with correct values to get the data. Moreover, AJAX calls can be connected to a specific user action, e.g., mouse move over a menu item, which can be hardly recognized and simulated by the extraction system. Additionally, some sort of servers may ask for registration, i.e., advanced work with cookies and POST and GET parameters is required.

Source Documents One of the basic problems during data retrieval from HTML documents is validity of the given document. One possibility is to transform the document to a valid XHTML [120] form which can be processed by common tools for XML processing like XPath [28] or XQuery [21] queries. Another possibility is to use more native tools to work with invalid HTML like *OXPath* [136] or one of many CSS[3] selector based libraries. These libraries commonly use various extension of the CSS language to cover larger area of constructs. Although these tools give us a huge amount of possibilities, sometimes it is necessary to get more concrete data hidden in one particular HTML field, e.g., element `div` with several data fields concatenated into one string. Thus the importance of additional post-processing is high.

For example, we want to extract data about a subcontract from two different fragments of a Web page (depicted in Figures 5.1(a) and 5.1(b)) and connect these data to the PCO. Both the fragments contain red labeled sections numbered 1 to 4 (related to PCO ontological properties as depicted in Table 5.1). Figure 5.1(a) depicts a fragment of TED[4] HTML document. Data are stored in elements `div` combined with additional textual information, i.e.,

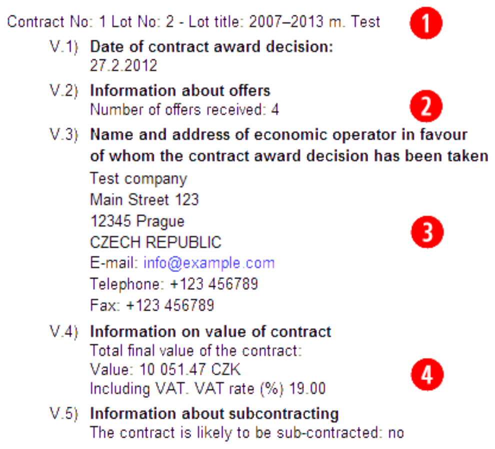
Contract No. 1 Lot No. 2 – Lot title: 2007-2013 m. Test

contains combined information about subcontract *id* (highlighted in blue), *name* (highlighted in red), and some unimportant labels. Thus it is necessary to extract these fields from the text, e.g., with the usage of regular expressions. Section 2 contains only one property:

Number of offers received: 4

with textual *label* “Number of offers received” which has to be removed. Similarly, in Sections 3 and 4 the fields are separated by elements `br` combined with additional labels, e.g., “E-mail”, “Telephone”, etc. On the other hand the data in ISVZUS[5] are strictly structured by elements `input` with unique attributes `id` (see Figure 5.1(b) for sections 1 to 4), thus it is possible to access the data fields directly without any additional transformation.

[2]Asynchronous JavaScript calls to insert dynamic content of a Web page
[3]Cascading Style Sheets (`http://www.w3.org/Style/CSS/`)
[4]Tenders Electronic Daily – European public procurement system (`http://ted.europa.eu/`)
[5]Czech national portal of public contracts (`http://www.isvzus.cz/`)


Contract No: 1 Lot No: 2 - Lot title: 2007–2013 m. Test 1

V.1) **Date of contract award decision:**
27.2.2012

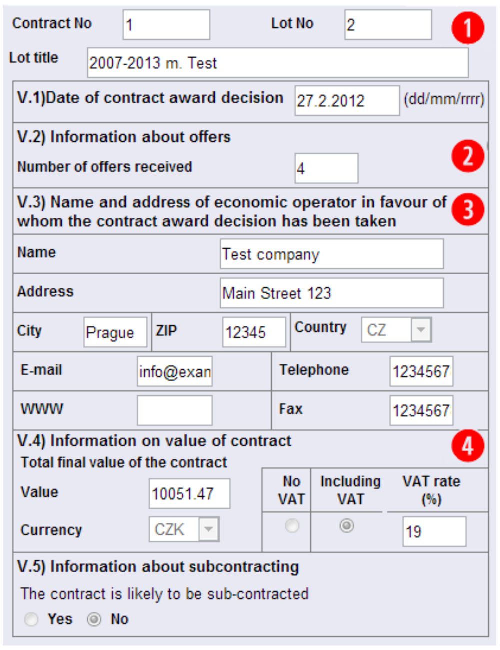
V.2) **Information about offers**
Number of offers received: 4 2

V.3) **Name and address of economic operator in favour of whom the contract award decision has been taken**
Test company
Main Street 123
12345 Prague
CZECH REPUBLIC 3
E-mail: info@example.com
Telephone: +123 456789
Fax: +123 456789

V.4) **Information on value of contract**
Total final value of the contract:
Value: 10 051.47 CZK 4
Including VAT. VAT rate (%) 19.00

V.5) **Information about subcontracting**
The contract is likely to be sub-contracted: no

(a) Text HTML Document



Contract No	1	Lot No	2
Lot title	2007-2013 m. Test		

1

V.1)Date of contract award decision 27.2.2012 (dd/mm/rrrr)

V.2) Information about offers 2
Number of offers received 4

V.3) Name and address of economic operator in favour of whom the contract award decision has been taken 3

Name	Test company				
Address	Main Street 123				
City	Prague	ZIP	12345	Country	CZ
E-mail	info@exan	Telephone	1234567		
WWW		Fax	1234567		

V.4) Information on value of contract 4
Total final value of the contract

		No VAT	Including VAT	VAT rate (%)
Value	10051.47			
Currency	CZK		✓	19

V.5) Information about subcontracting
The contract is likely to be sub-contracted
Yes [✓ No]

(b) Structured HTML Document

Figure 5.1: Web page fragments

#	PCO property	#	PCO property
1	dc:title + adms:identifier	3	pc:offeredPrice
2	pc:supplier	4	pc:numberOfTenders

Table 5.1: Mapping of PCO properties to HTML fragments from Figure 5.1

5.3 Related Work

Currently there exist several different approaches that use various ways to identify and extract data from documents in different formats. In this section we categorize them by the criteria presented in paper [40].

Level of Automation The extraction system can be fully automated with features to adapt to different aspects of the source documents. Other possible approaches are based on an input from the user. It can comprise wrappers, scripts or templates to be used for different documents. These system are based on hard-coded programs, or simpler scripts which have exact instructions how to process input documents. The main advantage of these solutions is their wide area of possible targets. The user can implement any application to gather the data from a source and export them into any format or send them to any other service. The problem is that the user has to know the programming or the script language. For example, paper [40] offers a high level of automation (it generates extracting wrapper for any given HTML document), but it has several requirements for the page, especially in the sense of a stable layout of the page and visual availability of query results. In paper [41], the authors focus on localization of data-rich regions and they extract the relevant attribute-value pairs of records from Web pages across different sites. The method is based on an observation that labels and values usually occur near each other.

Object of Extraction The object of extraction can be, e.g., a structured part of the document, simple text data, or any multimedia files, i.e., images. For example, in paper [137] the authors focus on relational data in the form of a list. Firstly, they get a sample lists from multiple sources and use them to compare column splits. Based on these comparisons an extraction score that reflects the confidence in the table's quality, incorrect splits and bad alignments, is identified. On the other hand, the idea of *WebTables* [138] is focused on online table querying and it uses new techniques to search keywords over a corpus of tables based on hand-written detectors and statistically-trained classifiers. Similarly, the *xCrawl* [139] is a method exploiting the navigational structures of Web sites such as hierarchies, lists, or maps. It was inspired by the requirements of extraction of a product and a services description.

Domain Specific Solutions The scraping method can be either focused on a domain described by an ontology, or it is a universal method extracting and indexing any kind of data. The *Ontology-Assisted Data Extraction* (ODE) [42] is a system for automatic identifying of lists of data on a Web page. The resulting fragments are then compared with an ontology and the data and their labels are assigned (based on the maximum correlation). On the other hand, *DIADEM* [140] is focused on the low level of Web pattern recognition, which uses machine-learning methods and linguistic analysis with basic ontology annotation. Additionally it uses goal-directed domain specific rules in order to identify the main extraction data structure and to finalize the navigation process. The system uses the *OXPath* [136] as a wrapper language, which is an extension of XPath with a support for user actions, i.e., simulation of mouse move, mouse clicks, document loading, etc.

Additional Aspects Another group of data extractors are publicly available or commercial solutions focused on different additional aspects. For example the *ScraperWiki* [141] is based on the community support. It offers automatic scheduled execution of the script. Each script is public and it is available for any user. Many of these solutions use nice user interface (e.g., the *Visual Web Ripper* [142]) and they are more or less denoted to users without programming skills as they allow the user to specify the object of scraping only through a simple interface and simple parameters.

Transformational Languages Last but not least, we have to describe transformational languages which can in composition with any simple crawler offer similar functionality as the scraper itself. One of the most obvious solutions (as far as we are in the world of Web and the most of the documents we are working with are HTML or XHTML) are languages like XSLT [69] or XQuery [21]. These languages offer high possibilities while manipulating with XML based files with possible output in RDF/XML [143]. Another possible approach to manipulate with text documents is usage of regular expressions to simply extract and transform the data.

5.3.1 Work in Context of Our Previous Work

Although the data acquired by *Strigil* are published in an open format, so that it is possible to use them in different systems which can work with RDF triples, we designed it as a part of three-level framework to extract, transform, and visualize the data. The overall architecture is depicted in Figure 5.2. The framework consists of three independent modules, i.e.,

1. *Strigil* – The extract core of the framework is used to acquire the data from different document sources, e.g., HTML, XLS, etc. This framework is described in the rest of this section.

2. *ODCleanStore* [144] – In this module, the converted data are cleaned, linked to other stored data and data in the linked open data cloud[6]. *ODCleanStore* provides aggregated views on the requested data. Furthermore, it accompanies all the aggregated data with descriptive and provenance metadata (e.g., the creation date, the process responsible for creating the data, the data license, source of the data) and information about the quality of the data to help consumers to decide which data are worth using.

3. *Payola*[7] – This visualization and analytical module simplifies work of application developers and users by enabling them to determine which linked data returned by *ODCleanStore* should be visualized, how, and whether an analysis of the data should be conducted (e.g., computing the average salary in a company).

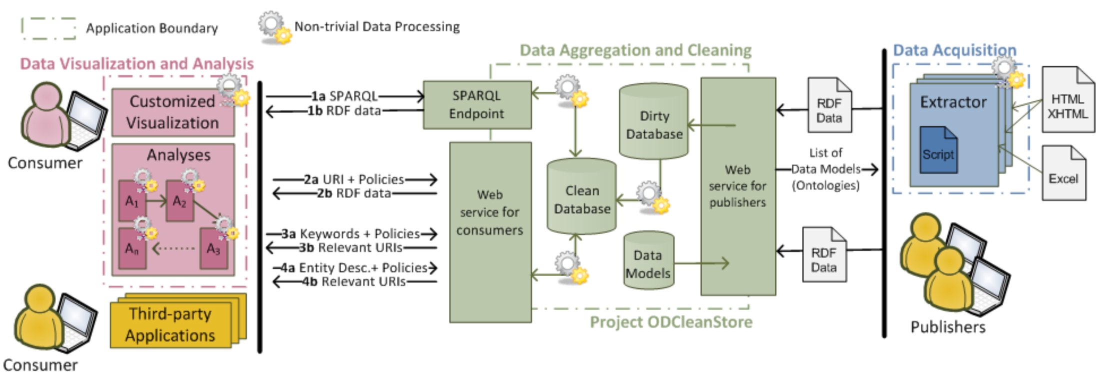


Figure 5.2: Linked Data projects architecture

5.4 Architecture of *Strigil*

From the highest level of perspective *Strigil* is divided into three logical parts (depicted in Figure 5.3):

- *Web application* is responsible for user inputs, i.e., it saves scraping scripts and various user-defined settings to the database.

- *Data application* (DA) controls the scraping process and provides the resulting RDF triples to *ODCleanStore* or saves them to a file. This process consists of several components and communicates with other *Strigil* modules. All the major components implement given interfaces through which the Data Application can be easily extended and extra functionality added.

[6]`http://linkeddata.org/`
[7]`http://payola.github.io/Payola/`

- *Download system* (DS) receives download requests from the DA and manages and plans them in order to keep user-defined politeness policy restrictions.

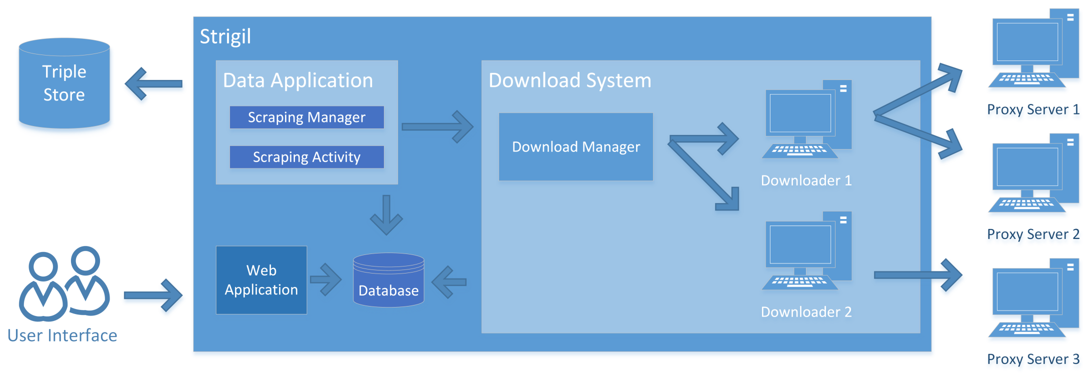


Figure 5.3: *Strigil* architecture

All these parts are completely separated components which may even run on different machines as they communicate through JMS[8]. The DS and DA are divided into even more separated components, described in the following paragraphs.

5.4.1 Download System (DS)

As mentioned above, the DS is responsible for managing, planning and executing download requests. It may be scaled based on user's needs. This scaling is done by multiplying some components and run them on different machines. It consists of three main components: the *Download Manager* (DM), *downloader(s)* and *proxy(ies)*. Each of these components may run on a different machine.

The DM receives download requests directly from the DA, which are then planned. The planning is done dynamically based on the downloaders and proxies available and it maximizes the performance with respects to the politeness policy defined. The download requests are planned for one specific downloader through one specific proxy server. If the number of download requests for one domain exceed the maximum allowed quota per minute, the download requests are delayed until the quota is freed. In other words, the DM keeps user-defined politeness policies.

A downloader receives particular download requests for a document and uses a proxy to download it. It maintains connections to all available proxy servers. These components may be multiplied to provide the download performance the user requires. Increasing the number of downloaders directly increases the download throughput of the whole system. Since Web servers usually have very restrictive limitations of downloads per hour or per day, it is useful to provide a way to access the data sources from more computers (IP addresses). And that is exactly how the proxy servers are used.

5.4.2 Data Application (DA)

The major components ensuring DA functionality are: the *scraping manager* with a *scheduler*, *scraping activity* and *URL request queue controller* (URQ). They communicate mutually and cooperate during the scraping process. The scraping manager

[8] `http://en.wikipedia.org/wiki/Java_Message_Service`

assigns tasks to other components, keeps statistics, and processes results. It creates scraping activities responsible for parsing actual data from a given source according to a script and collects their results or requests for additional document downloads. Download requests are pushed to URQ which sends them to the DS together with callback to be executed on a successful download or a failure. Finally, the scraping manager composes the results and returns them to a triple store or saves to a file.

The DA is designed to be easily extended with new features, e.g., parsing other types of documents, using different sources of data or sending the results to other consumers. The architecture of DA is depicted in Figure 5.4.

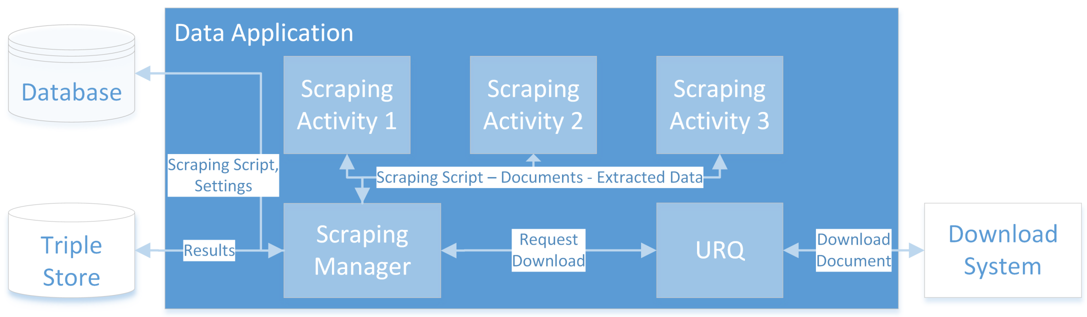


Figure 5.4: Data application architecture

Scraping Manager The scraping manager contains the main logic of the DA. It gets from the database information about scripts to run and basically starts, controls and finishes the entire scraping process. It creates and submits scraping activities, provides them data, stores their partial results, handles their failures, etc. It also calls methods on the URQ controller to create and push download requests to the DM. Last but not least, the scraping manager provides results, i.e., writes RDF triples to files of a filesystem and/or sends them to the *ODCleanStore triplestore*. Additionally it stores statistics about scraping process.

The implementation of scraping manager uses the *Quartz Scheduler framework*[9] to implement scheduling of script execution. This framework works with the concept of *jobs* and *triggers*. Triggers define when or how often the jobs will be executed. One job can be assigned with many triggers. Quartz allows many properties to be configured for a scheduler, e.g., the scheduler instance name, the number of threads for executing jobs, or, one of the most important ones, where the working data is kept – in the RAM or a database. If a database is used, Quartz is distributed with necessary create-tables scripts for the majority of database vendors. Note that these structures should never be accessed directly. It is strongly recommended to always use the provided API.

Strigil uses two instances of Quartz scheduler. Both store working data in a database and on top of that they are pointed to the same database tables. This allows both instances to share the jobs and triggers. One instance runs in the Web application, where users can schedule available scripts, but the scheduler is not actually started. In other words, the Web application just stores the triggers. The second instance is started in the scraping application where jobs are created and started. The scheduler also keeps track of misfired triggers (e.g., when the scheduler is stopped, the DA is not running etc.) and fires them as soon as possible.

[9]http://quartz-scheduler.org

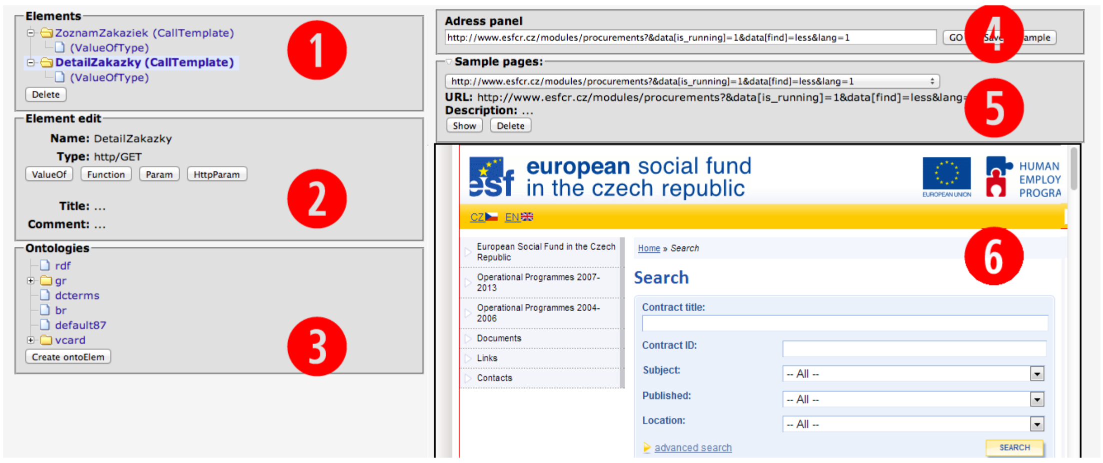
Scraping Activity The scraping activity is the part of *Strigil* that performs data extraction from documents, and afterwards creates the resulting RDF graphs for individual documents. In order to do this, the activity needs to have the information how to download documents, which data to extract from these documents, and it needs to know about the ontological relations between extracted data. This information is given to activity in the form of *scraping script*. In other words, the activity receives a scraping script as input, and then it executes the script. Every activity runs in its own thread, so it is possible to run multiple activities in parallel to increase overall performance. The activity returns the extracted data and a *coverage* of used selectors, i.e. it compares the number of used selectors and the number of returned empty responses. The system notifies the user if the ratio is bellow a user-defined threshold (the script could be wrong or the source documents could have been changed).

5.4.3 Web Application

The Web application provides an administration user interface which allows management of scraping scripts, proxy servers, downloaders and overall policies. Additionally, it contains a Web editor for scraping script creation with HTML page inspector, intelligent selection of CSS selectors and support for intelligent ontology property assignment, i.e., a user can import custom ontologies and then the user interface offers available ontological properties.

This module also provides a Web Service to store scripts from third-party editors. For example, an implementation of MS Excel selectors can directly communicate with *Strigil* from an add-on module.

In Figure 5.5 the main parts of the HTML script editor are depicted. The editor consists of six panels. Panel 1 shows the tree structure of edited script, i.e., the ontological elements, called functions, used selectors, etc. On Panel 2 the properties of the currently edited script elements are displayed. The selectors can be acquired by selecting particular elements in Panel 6 which displays a sample script source document. Panel 3 displays available ontologies used for intelligent offering of new data mappings. Panels 4 and 5 are used to browse sample documents in an inline browser.


Elements
ZoznamZakaziek (CallTemplate)
(ValueOfType)
DetailZakazky (CallTemplate)
(ValueOfType)

Element edit
Name: DetailZakazky
Type: http/GET
Title: ...
Comment: ...

Ontologies
rdf
gr
dcterms
br
default87
vcard

Adress panel
http://www.esfcr.cz/modules/procurements?&data[is_running]=1&data[find]=less&lang=1

Sample pages:
http://www.esfcr.cz/modules/procurements?&data[is_running]=1&data[find]=less&lang=1
URL: http://www.esfcr.cz/modules/procurements?&data[is_running]=1&data[find]=less&lang=
Description: ...

Contract title:
Contract ID:
Subject: -- All --
Published: -- All --
Location: -- All --

Figure 5.5: Script editor GUI for HTML documents

5.5 Scraping Script

The scraping script is basically an XML document, which contains instructions telling the *Strigil* system how to obtain input documents. It also describes the way how to handle them – how to extract relevant data from documents, process them (e.g., convert to a suitable format, like date or decimal number), and map them to ontological classes and attributes.

The result of a processed script is an RDF graph or a set of all semantically valid RDF triples that contain data extracted from all documents, obtained during the script runtime. Usually that means all "interesting" documents from one specific server for which the script is written.

Since the script is essentially an XML document, it is possible to create and edit scripts in any XML or text editor. However, it is possible to use the script editor built-in Web application interface or an external tool. It is not only more comfortable, but it also disallows the user to create a script, which is not valid against the script schema[10].

In *Strigil*, we decided to design our own script language, inspired by XSLT [69], to transform different documents to data specified by ontologies. The script language uses *selectors* to specify data fields in a source document. Although we describe selectors for HTML documents, the selectors can be used to extract data from other formats depending on available implementations. Currently we work with *JSoup* selectors[11] for HTML documents and our custom selectors for Excel files (similar to HTML selectors, extended by some constructs to work with work sheets). Despite our selection, it is possible to use any other selectors like, e.g., *OXPath* [136] to get data.

The script has one root element `script` with four parts – represented by elements `meta`, `params`, `template`, and `call-template` – described in the following paragraphs. All parts, except for element `params`, are required. The complete structure of the script is validated against the respective XSD[12]. A fragment of the schema, which defines the element `script`, is provided in Listing 5.1.

Element `script` has the following attributes:

- `prefix` (declaration of namespaces of ontologies used within the script),
- `id` (unique identifier of the script),
- `version` (number representation of script version), and
- `type` (type of the script, e.g., HTML, Excel, etc.).

Element `meta` Element `meta` is used to store important details about the script, i.e., author of the script, date of the last modification, the domain, for which the script is written and the status of the script (i.e., draft or released). This element has no child elements.

[10]`http://sourceforge.net/p/strigil/code/HEAD/tree/doc/Scripts/Schema_27_11_2012/Schema_27-11-2012.xsd`

[11]`http://jsoup.org/cookbook/extracting-data/selector-syntax`

[12]`http://sourceforge.net/p/strigil/code/HEAD/tree/doc/Scripts/Schema_27_11_2012/Schema_27-11-2012.xsd`

```
<xs:element name="script">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="call-template" type="call-
         templateType"/>
      <xs:element name="meta" type="metaType"/>
      <xs:element name="params" type="paramsType"
         minOccurs="0"/>
      <xs:element name="template" type="templateType"
         maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:NCName" use="
       required"/>
    <xs:attribute name="prefix" type="lst_string" use="
       optional" />
    <xs:attribute name="version" type="xs:string" use="
       optional"/>
    <xs:attribute name="type" type="document_type" use="
       required" />
  </xs:complexType>
</xs:element>
```

Listing 5.1: XML Schema definition of element `script`

Element `params` Element `params` is optional, and it is called *script parameters declaration*. In this part, the author of the script can declare parameters which are available inside the script. Their value is usually specified when the script is called for execution, but a default value can also be specified during declaration. It allows customization of script behavior, e.g., it allows the user to run the script only for some specific URL addresses, which can be different on every run.

Parameters are defined inside element `params` using element `param`, one for each parameter. Names of parameters are defined inside attribute `name`, default value can be specified either in attribute `default`, or inside element‘s body. If both default values are found, default attribute has a higher priority.

Element `template` Templates are the main executive part of the script. A template describes the way document is processed, how useful information is extracted from source documents, and how this information is mapped to ontology classes and properties. Each template can contain several ontological elements representing instances of ontology classes. Each of these classes can contain several properties extracted from the source document by selectors. The selectors depend on the type of the document. Currently, we work with HTML documents and work sheet documents (MS Excel). For the HTML documents we use extended CSS selectors (the *JSoup* library) which allow the user to find the elements on the basis of common HTML attributes like `id`, `name`, or `class`. Additionally, it allows the user to find elements with specified text content or elements valid against a regular expression.

In case of XLS documents we designed our own set of selectors working with the lists and columns in a similar way. The selectors allow the user to select columns based on their position, their content or their relative position against other located column.

Element `call-template` In almost every script, it is possible to find more than one template. For example, we can consider a script, that will extract details about top 250 movies from the *IMDb*[13] . In this script, we need a template that will parse an HTML document with the list of movies (and URL addresses of documents with details about movies) and a template which will contain rules for data extraction from the document with details about the movie (see Figure 5.6). Then, we need to call a second template from the first template, to get URL addresses of all movies. The template call is realized by element `call-template`. The name of template which will be called is specified in its attribute `name`, and the URLs of documents to parse with this template, are specified in its body using elements `value-of`.

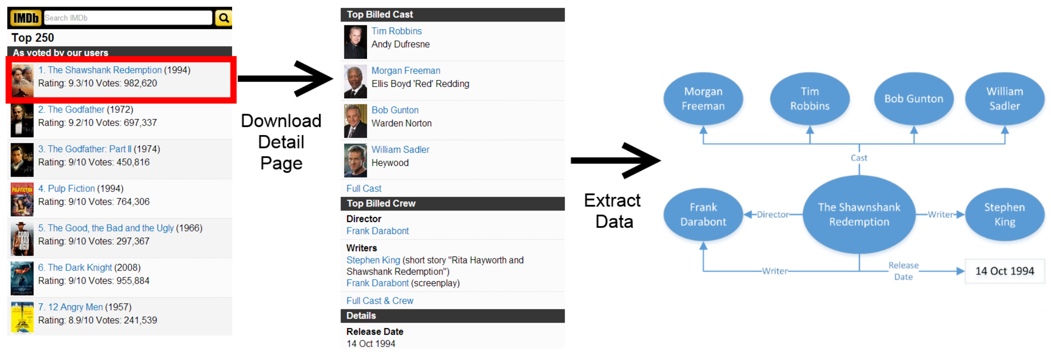


Figure 5.6: *Strigil* – template processing example

If we want to call a template on documents downloaded by the HTTP POST method, we can also specify HTTP POST parameters. This is useful for example if we want to get a document hidden behind a HTML form. HTTP POST parameters are added to the call using element `http-params`, which is a descendant of element `call-template`. The parameters are passed in child elements `with-param` of element `http-param`. The name of the parameter is inside attribute `name`, or, if the name of parameter is stored in variable, we can specify the name with attribute `namevar`. The value is stored inside element‘s content.

Besides the HTTP POST parameters, it is also possible to define cookies for the HTTP communication with a source server. This is useful, e.g., when server sets the language of documents based on a user’s choice, and the chosen language is saved as a cookie. All cookies, used during HTTP communication (and also cookies newly set by server), are saved in the template context, and they are used for every descendant template called from the current template. cookies are also set inside element `http-params`, but element `cookie` is then used instead of element `with-param`.

In the following paragraphs we describe parts of the script responsible for ontology mapping (element `onto-elem`), data extraction (element `value-of`), and data processing (variables and functions).

[13]Internet Movie Database (`http://www.imdb.com`)

Element `onto-elem` Element `onto-elem` is used to create a new RDF resource. All properties added to an RDF graph in the body of this element will be added as properties of this resource. Also, all other resources created in the body of this element (its direct descendants) must have attribute `rel` which adds relation between this element and its descendants. This element can contain the following attributes:

- `rel` (relation to a parent resource, if the parent of this element is also element `onto-elem`),
- `typeof` (value of property `rdf:type` of the resource),
- `about` (URI of the resource).

Element `value-of` Text values in the script need to be defined using elements `value-of`. The value is returned from the element after evaluation of its attributes. Attributes are evaluated in the following order (an attribute is evaluated, only if the preceding attribute is not found or empty):

- Text constant – When attribute `text` is present, its value is directly returned as a value of the element.
- Variable – When attribute `var` is present, it resolves the value of the variable with the name specified in this attribute, and that value is returned as a value of the element.
- Replacement – When the attributes `select`, `regexp` and `replace` are present, and attributes `select` and `regexp` have a non-empty value, and matching groups are used in the regular expression, attribute `replace` is used to build the resulting string.
- Regular expression – When attributes `select` and `regexp` are present, and attribute `select` is not null, the value is checked. If it satisfies the regular expression defined in attribute `regexp`, the value is returned, otherwise an empty string is returned.
- Selector value – When attribute `select` is present, the selector (e.g., CSS) in the attribute is evaluated against the source document and the selected value is returned.

In Listing 5.2 the content of a template called *Detail* is shown. It works with a tender detail to get a basic information about tender title and tender deadline. For these purposes element `value-of` is used to get data with the given selector. Additionally, function `convert-date` is used to extract a date in some predefined format from the string.

Variables Naturally, it is also possible to use variables within templates. They can hold values, which can be either known to the script author and assigned during the script creation phase, or they can be assigned with values determined during the script run, e.g., as a result of a function, or a value extracted from a source document. The type of a variable is an array of strings. A variable can only be seen in the template, where it is declared.

```
<scr:onto-elem>
  <scr:onto-elem
    rel="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
    about="http://purl.org/procurement/public-contracts#
       Contract" />
    <scr:value-of
      select="body#esf␣div#article␣h2"
      property="http://purl.org/dc/elements/1.1/title" />
    <scr:function name="convertDate"
      property="http://purl.org/procurement/public-
         contracts#tenderDeadline">
      <scr:with-param>
        <scr:value-of
          select="body#esf␣div#wrapper␣p:contains(Example
             ␣text)" />
        </scr:with-param>
    </scr:function>
</scr:onto-elem>
```

Listing 5.2: An example of element `onto-elem`

To declare a variable we use element `variables` at the beginning of the template definition. Inside this element, we declare all used variables with elements `variable`, one element for each variable. Variable name is specified by its attribute `name`.

Assigning a value to a variable is done via element `assign`. This element receives the name of a variable inside its attribute `var`. The value, which is assigned to this variable is evaluated from the descendant elements. If the value is already known, inside other variable, or can be discovered by simple extraction, we use the descendant `value-of`. If it is necessary to use function(s) to receive the value, we use the descendant `function`.

To get a value from a variable, we use element `value-of` with attribute `var`. Inside this attribute is the `name` of the variable, whose value we want to receive. If the element `value-of` has attribute `property`, the value of this variable will be added to the output RDF graph as the object of the predicate specified by attribute `property`.

Functions In some cases, there is a need for a modification of (not only) extracted data, e.g., conversion to a valid data format, concatenating strings to create URLs, etc. To help with these situations *Strigil* has a built-in set of implemented functions that can be called from the script. A function call is created by element `function`; in attribute `name` the name of the function is specified. It is also possible to add parameters to these functions via the element `with-param`.

The list of available functions is as follows:

- `conc` – This function returns the string created by concatenation of the strings received in the parameters.

- `concatenate` – This function returns the array of string created by appending the value of the first parameter (prefix) to every string from the second parameter (suffixes).
- `convertDate` – This function converts the date from formats “`DD.MM.YYYY`”, “`MM/DD/YYYY`”, or “`DD. MMMM YYYY`” to XML Schema date format (i.e., “`YYYY-MM-DD`”). The name of month must be in English or Czech language, to convert properly.
- `convertSpacesToHtmlEntity` – This function takes the input string as an argument, and converts all white space characters to HTML space representation (i.e., a sequence of “`%20`”).
- `convertToFloat` – This function converts a float string to the valid XML Schema float type.
- `generateUUID` – This function returns randomly generated UUID[14] which can be used to create random strings, e.g., unique URIs for elements `onto-elem`.
- `getCurrentUrl` – This function returns a URL of the currently processed document.
- `removeSpaces` – This function removes white space characters from the input string.

In some cases, there is a need to execute another script, from the currently executed script. Let us have a situation, that we are creating a script, which will extract the data from HTML documents containing information about public contracts. Between these information, we can find only a name of the winning company, and its identification number. We want to get as much information about the company, as possible. These information can be found on the business register Web site. As a solution, we can prepare two different scripts. One for the Web site containing details about public contracts, and one for the business register. In the first script (public contracts), we insert a script call, which will start the business register script, and inside this call we give the identification number of the company, that won the contract. The business register script will then extract the information about the company.

Parameters used inside a script call are optional. They are specified using element `with-param`. Their name must be declared inside the called script, otherwise the parameter will be ignored.

The script call is realized using element `call-script` inside the template.

In Listing 5.3, there are three template calls. The first one is the entry template, called directly after the script starts. The element contains attributes for its name and the download method. Additionally it contains element `value-of` which contains a fixed value for the input document. The second template called *List* expects on input a list of procurements. We want to browse them and for each its detail we want to run the third template. Thus there is element `call-template` called on a CSS selector to get the URL of all list anchors. A part of the result from Listing 5.3 (and Listing 5.2 which describes the content of template *Detail*) is show in Listing 5.4.

[14]Universally unique identifier

```
<scr:call-template
      name="List"
      type="http/GET">
   <scr:value-of
      text="http://www.ex.cz/data/list/" />
</scr:call-template>
<scr:template
      name="List"
      mime="text/html">
   <scr:samplePage
      url="http://www.ex.cz/data/ist/" />
      <scr:call-template
            name="Detail"
            type="http/GET">
         <scr:value-of
            select="body#esf␣div#main␣a␣@href"
         />
   </scr:call-template>
</scr:template>
```

Listing 5.3: `call-template` example

```
<rdf:RDF ... >
 <rdf:Description rdf:nodeID="A0">
      <dc:title>Sample tender</dc:title>
      <pc:tenderDeadline
         rdf:datatype="http://www.w3.org/2001/XMLSchema#
            date">
           2013-03-29
      </pc:tenderDeadline>
      <rdf:type
         rdf:resource="http://purl.org/procurement/public-
            contracts#Contract"/>
 </rdf:Description>
 <rdf:Description rdf:nodeID="A1">
      <dc:title>Another tender</dc:title>
      <pc:tenderDeadline
         rdf:datatype="http://www.w3.org/2001/XMLSchema#
            date">
           2013-04-14
      </pc:tenderDeadline>
           ...
```

Listing 5.4: Scraping activity output

Source	Documents	Triples	Average Number of Triples per Document
TED	5693	21884	3.8
ESFCR	688	23288	33.5

Table 5.2: Experimental data sources

Summary In comparison to other scraping or transformation languages, our approach provides with following advantages:

- Ontology support – Although many scripts languages are able to generate RDF triples defining the correct output in the wrapper, *Strigil* scraping language offers an explicit support for the classes and properties through element `onto-elem` and attribute `property`. It increases simplicity and readability of the language.
- Download support – The language is designed with regard to the data scraping, i.e., it supports navigation through the Web pages, usage of HTML forms, cookies, etc.
- Customizable selectors – The structure of the script is ontology-oriented. It means that the system can use any implemented selectors to extract data from different formats with minimal changes to the structure of the script.

5.6 Experiments

To demonstrate the possibilities of *Strigil* and the proposed scraping language we have prepared two scripts. Both of them extract data about public procurements and map them to the PCO. Firstly, we extracted the published data about European procurements from TED from August 30, 2010 to September 8, 2010 to demonstrate usage of HTML forms. Secondly, we prepared a complex scraping script to get all procurement documents from the European Social Fund in the Czech Republic (ESFCR)[15]. The script extracts detailed data about contracts, i.e. title, publication date, estimated price, location, tender deadline, or address of the contracting authority, and we show the possibilities of data extraction by CSS selectors combined with regular expressions, and functions (to convert dates and numbers). Overall information about the number of documents and extracted triples is shown in Table 5.2. Both scripts are publicly available on the Web[16].

5.7 Conclusion

In this section, we have presented *Strigil*, a system for automatic data extraction, which is a part of a framework for data processing and integration. The aim of this system is to extract data from one domain based on user-defined scripts. We presented the main components of the framework and their main advantages. We described our scraping

[15]`http://www.esfcr.cz/`

[16]`http://ksi.mff.cuni.cz/~starka/strigil_scripts.zip`

language and its main features focused on Web crawling, ontology integration and data extraction.

Although the work in this research area is wide, there are still several aspects to improve. Firstly, the structure of HTML documents describing the same entity is changing. Currently most of the Web pages contain content that is dynamically generated and thus the acquired documents can contain differently rendered parts, such as, e.g. commercials, included date, last news, etc. This invokes the necessity for scraping scripts to be general and to allow changes in the structure of the input documents. Moreover, the server can change the layout entirely, thus it is important to be able to find out these changes as soon as possible and inform about this change or improve the script automatically.

In our future work we will focus on automatic script generation and adaptability to the design changes. Despite having a universal CSS selector, there are still areas of extraction to improve, e.g., the combined text outputs need complex regular expressions. Additionally, we will focus on extensions of supported formats like CSV, or the possibility to get the data from different sources, such as, e.g., Web Services, database dumps, etc.

6. Schema Inference

One of the main research areas which widely exploits the knowledge of complexity of real-world data provided by Analyzer *is inference of XML schemas from a given sample set of XML documents. As we already mentioned in Section 1, various statistical analyses of real-world XML data show that a significant portion of XML documents (in particular 52% [6] of randomly crawled or 7.4% [14] of semi-automatically collected) still have no schema at all. What is more, XML Schema definitions (XSDs) are used even less (only for 0.09% [6] of randomly crawled or 38% [14] of semi-automatically collected XML documents). Thus a research area of automatic construction of an XML schema has opened. The key aim is to create an XML schema for the given sample set of XML documents that is neither too general, nor too restrictive.*

In this section we introduce jInfer, *a general framework for XML schema inference. It represents and easily extensible tool that enables one to implement, test and compare new modules of the inference process. Since the compulsory parts of the process, such as parsing of XML data, visualization of automata, transformation of automata to XML schemas etc. are implemented, the user can focus purely on the research area and the improved aspect of the inference process. We describe not only the framework, but the area of schema inference in general, including related work and open problems. We also describe three of our improvements and extensions of the current state of the art.*

The content of this section corresponds to a journal paper jInfer: a Framework for XML Schema Inference *[63] that was submitted to the Computer Journal (IF: 0.785, 5-Year IF: 0.943), and it is currently under a review process. This article extends our previous published papers:* Optimization and Refinement of XML Schema Inference Approaches *[56] published at the 3rd International Conference on Ambient Systems, Networks and Technologies (ANT 2012),* Inference of an XML Schema with the Knowledge of XML Operations *[57] published at the 8th International Conference on Signal Image Technology and Internet Based Systems (SITIS 2012), and* Schematron Schema Inference *[55] published at the 16th International Database Engineering & Applications Symposium (IDEAS 2012).*

6.1 Introduction

Statistical analyses of real-world XML data show that a significant portion of XML documents (52% [6] of randomly crawled or 7.4% [14] of semi-automatically collected) still have no schema at all. What is more, XML Schema definitions (XSDs) are used even less (only for 0.09% [6] of randomly crawled or 38% [14] of semi-automatically collected XML documents) and even if they are used, they often (in 85% of cases [5]) define so-called *local tree grammars* [81], i.e., grammars that can be defined using DTD as well.

Consequently a new research area of automatic construction of an XML schema has opened. The key aim is to create an XML schema for the given sample set of XML documents that is neither too general, nor too restrictive. It means that the set of document instances of the inferred schema is not too broad in comparison with the sample data but, also, it is not equivalent to it. Currently, there are several proposals of respective algorithms, but there is also still a space for further improvements. In particular, since according to Gold's theorem [11] regular languages are not identifiable only from positive examples (i.e., sample XML documents which should be valid against to the resulting schema), the existing methods need to exploit either heuristics or a restriction to an *identifiable* subclass of regular languages.

Contributions In this paper we introduce *jInfer*, a general framework for XML schema inference. It represents an easily extensible tool that enables one to implement, test and compare new modules of the inference process. Since the compulsory parts of the process, such as parsing of XML data, visualization of automata, transformation of automata to XML schemas etc. are implemented, the user can focus purely on the research area and the improved aspect of the inference process. We describe not only the framework, but the area of schema inference in general, including related work and open problems. Note that a similar system, called *SchemaScope*, was described in [44]; however, its main target are grammar-inferring approaches, especially those proposed by its author. In *jInfer* we focus on more general view of the problem, involving mainly the heuristic approaches.

6.2 XML Schema Languages

Nowadays, there exist several languages for description of an XML schema, i.e., the allowed structure of XML documents. The best known and most commonly used representatives are DTD, XML Schema, RELAX NG, and Schematron.

DTD The simplest and most popular language for description of the allowed structure of XML documents is currently the *Document Type Definition* (*DTD*) [1]. It enables one to specify allowed elements, attributes and their mutual relationships, order and number of occurrences of subelements (using operators '`,`', '`|`', '`?`', '`+`' and '`*`'), data types (`ID`, `IDREF`, `IDREFS`, `CDATA` or `PCDATA`) and allowed occurrences of attributes (`IMPLIED`, `REQUIRED` or `FIXED`). A simple example of a database of employees is depicted in Figure 6.1.

At first glance it seems that the specification of the allowed structure is sufficient. Nevertheless, even in this simple example we can find several problems. For instance, we are not able to specify the correct structure of an e-mail address. Similarly, we cannot simply specify that a person can have four e-mail addresses at maximum. And, as we can see, the fact that the order of elements `first` and `surname` is not significant cannot be expressed simply as well.

XML Schema With regard to the insufficiency of DTD, the W3C[1] proposed a more powerful tool – *XML Schema* [23, 68] and its instances called *XML Schema Definitions* (*XSDs*). An example of an XSD equivalent to the example of a DTD in Figure 6.1 is depicted in Figure 6.4. XML Schema involves all the DTD constructs, only the syntax is different (e.g., element `sequence` represents operator '`,`', `choice` represents '`|`', content models are specified using `simpleTypes` and `complexTypes`). It also adds several new constructs that can be divided into "syntactic sugar" and true new constructs extending the expressive power. The former class involves, e.g., precise occurrence ranges (i.e., attributes `minOccurs` and `maxOccurs`) or globally defined

[1]`http://www.w3.org/`

```
<!ELEMENT employees (person)+>
<!ELEMENT person (name, email*, relationships?)>
  <!ATTLIST person id ID #REQUIRED>
  <!ATTLIST person note CDATA #IMPLIED>
  <!ATTLIST person holiday (yes|no) "no">
<!ELEMENT name ((first, surname)|(surname, first))>
<!ELEMENT first (#PCDATA)>
<!ELEMENT surname (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT relationships EMPTY>
  <!ATTLIST relationships superior IDREF #IMPLIED
                          inferior IDREFS #IMPLIED>
```

Figure 6.1: An example of a DTD

```
<element xmlns="http://relaxng.org/ns/structure/1.0"
         name="employees">
  <oneOrMore>
    <element name="person">
      <element name="name">
        <interleave>
          <element name="first"> <text/> </element>
          <element name="surname"> <text/> </element>
        </interleave>
      </element>
      …
      <attribute name="id">
        <data type="ID"/>
      </attribute>
      ...
    </element>
  </oneOrMore>
</element>
```

```
element employees {
  element person {
    element name {
      element first { text } &
      element surtname { text }
    } ,
    … ,
    attribute id { xs:ID } ,
    ...
  } +
}
```

Figure 6.2: An example of a RELAX NG schema

```
<schema xmlns="http://www.ascc.net/xml/schematron">
  <pattern name="Conditions for list of employees">
    <rule context="employees">
      <assert test="person">No person found</assert>
    </rule>
  </pattern>
  <pattern name="Conditions for one person">
    <rule context="person">
      <assert test="name">A person must have an
        element name</assert>
      <report test="name[2]">A person cannot have
        multiple elements name</report>
      <assert test="@id">A person must have an
        attribute id</assert>
      ...
    </rule>
  </pattern>
  ...
</schema>
```

Figure 6.3: An example of a Schematron schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="employees">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="person" minOccurs="1"
                    maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element name="email" type="xs:string"
                    minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="relationships" minOccurs="0"
                    maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:ID" use="required"/>
      <xs:attribute name="note" type="xs:string"/>
      <xs:attribute name="holiday" default="no">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="yes"/>
            <xs:enumeration value="no"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>

  <xs:element name="name">
    <xs:complexType>
      <xs:all>
        <xs:element name="first" type="xs:string"/>
        <xs:element name="surname" type="xs:string"/>
      </xs:all>
    </xs:complexType>
  </xs:element>

  <xs:element name="relationships">
    <xs:complexType>
      <xs:attribute name="superior" type="xs:IDREF"/>
      <xs:attribute name="inferior" type="xs:IDREFS"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 6.4: An example of an XSD

items (i.e., simple/complex types, elements/attributes, groups of elements/attributes). In the latter class we can find a new rich set of simple data types (such as `integer` or `date`), user-defined simple types, derivation of complex data types from existing ones, advanced identity constraints (i.e., `unique`, `key`, `keyref`) or assertions (i.e., `assert`, `report`) supported since version 1.1 [124, 125]. In addition, each XSD is an XML document, hence, for its processing we can exploit any XML technology.

On the other hand, XML Schema has also several disadvantages. Firstly, as we can see from the examples in Figure 6.1 and 6.4, the description of an XSD is much longer and less lucid than the respective DTD. In addition, since the language involves a huge amount of constructs, which are mostly a "syntactic sugar" and hence have the same or almost the same expressive power, it is not easy for a user to learn all of them and decide which is better to use.

RELAX NG The authors of *RELAX NG* [102] tried to propose a language that involves key advantages of both DTD and XML Schema, but avoids their disadvantages. In particular, as we can see in Figure 6.2 the language has both XML and compact syntax which are equivalent and mutually transferable, hence it exploits the advantages of both DTD and XML Schema. Contrary to DTD it allows us to specify the structure of a mixed-content element precisely (i.e., like XML Schema does); contrary to XML Schema it does not restrict the complexity of unordered sequences only to simple cases. Also, similarly to XML Schema, it supports a wide set of simple data types, both built-in and user-defined. On the other hand, precise occurrence ranges of elements or groups of elements are surprisingly not supported – RELAX NG supports the same operators as DTD (i.e., `optional`, `oneOrMore` and `zeroOrMore`). Last but not least, contrary to both DTD and XML Schema, the definition of a content model can combine both elements and attributes, hence the expressive power is much higher than in case of DTD. XML Schema can express similar restrictions using assertions, but these were established in version 1.1 and currently are not much supported so far.

Schematron Contrary to the previous languages where we can find numerous similarities, *Schematron* [51] uses a completely different strategy. However, on the other hand, it probably served as an inspiration for XML Schema assertions. As we can see in Figure 6.3, each Schematron schema is based on the idea of *patterns*. A pattern can be described as a set of *rules* an XML document must satisfy to be valid against a Schematron schema. A rule involves either element `assert` or `report` depending on the requirement of satisfaction or not satisfaction of the condition specified in attribute `test`. Since a Schematron schema can be evaluated by translation to XSLT [69] and usage of an XSLT parser, it supports the same subset of XPath [28] as XSLT.

Even though the expressive power of Schematron is apparently higher than in case of the previous three XML schema languages (if we omit XML Schema assertions), Schematron has also several disadvantages. As we can see in Figure 6.3, the biggest problem is complexity of expressing simple rules such as "an element e has an attribute a". Hence, Schematron should rather be considered as an extension of the previous languages that enables one to express complex *integrity constraints*. Both XML Schema and RELAX NG support Schematron subschemas.

6.3 Related Work

The existing solutions to the problem of automatic construction of an XML schema can be classified according to several criteria such as the type of the result or the way it is constructed.

Heuristic approaches [145, 146, 147] are based on experience with manual construction of schemas. Their results do not belong to any class of grammars and they are based on generalization of a trivial schema using a set of predefined heuristic rules, such as, e.g., "if there are more than three occurrences of an element, it is probable that it can occur arbitrary times". These techniques can be further divided into methods which generalize the trivial schema until a satisfactory solution is reached [145, 146, 82] and methods which generate a huge number of candidates and then choose the best one [147]. While in the first case the methods are threatened by a wrong step which can cause generation of a suboptimal schema, in the latter case they have to cope with space overhead and specify a reasonable function for evaluation quality of the candidates. A special type of heuristic methods are so-called *merging state algorithms* [146, 82]. They are based on the idea of searching a space of all possible generalizations, i.e., XML schemas, of the given XML documents represented using a prefix tree automaton. By merging its states they construct the sub-optimal solution. In fact, since the space is theoretically infinite, only a reasonable subspace of possible solutions is searched using various heuristics.

On the other hand, methods based on *inferring of a grammar* [47, 148, 149, 150, 12, 13, 45] exploit the theory of languages and grammars and thus ensure a certain degree of quality of the result. We can view an XML schema as a grammar and an XML document valid against the schema as a word generated by the grammar. Although grammars accepting XML documents are in general context-free [151], the problem can be reduced to inferring of a set of regular expressions, each for a single element (and its subelements). But, since according to Gold's theorem [11] regular languages are not identifiable only from positive examples (i.e., sample XML documents which should conform to the resulting schema), the existing methods exploit various other information such as, e.g., predefined maximum number of nodes of the target automaton, restriction to an identifiable subclass of regular languages etc.

Almost all of the published approaches focus on inference of DTD content models. So far we have identified only three approaches which focus directly on true XML Schema constructs. The first approach was published in [45]. The authors define a subclass of XSDs which can be learned from positive examples and focus especially on constructs which are used in real-world XML schemas. The second approach described in [82] is a sort of heuristic merging state algorithm and it focuses on handling elements with the same name but different structure and unordered sequences. The last one [152] focuses on inference of various XML Schema constructs using user interaction.

6.4 Theoretical View of the Problem

In general, we can divide the described schema languages into *grammar-based* (i.e., DTD, XML Schema, RELAX NG) and *pattern-based* (i.e., Schematron). The majority of current works deal with basic and most common structural specification of XML data that can be expressed using the grammar-based languages. In other words they do not deal with advanced integrity constraints that can be expressed using Schematron or XML Schema assertions. So, if not stated otherwise, we consider the same set.

The grammar-based XML schema languages we consider in the first parts of our text can be further classified according to their expressive power. We borrow and slightly modify for our purposes the definitions from [81]. Since the well-formedness of XML documents [1] ensures that the correct usage of start and end tags forms a tree structure of XML documents, we usually speak about *tree grammars*. In addition, since most of the current approaches do not consider attributes, because their inference can be considered as a special case of inference of elements having a text content, we omit them for simplicity too.

Definition 11. *An* XML tree *is a directed labeled tree* $t = (V,\ E,\ \Sigma,\ \Gamma,\ lab,\ v_{root})$*, where:*

- V *is a finite set of nodes,*
- $E \subseteq V \times V$ *is a set of edges,*
- Σ *is a finite set of element names,*
- Γ *is a finite set of text values,*
- $lab : V \to \Sigma \cup \Gamma$ *is a surjective function which assigns a label to each* $v \in V$*, and*
- $v_{root} \in V$ *is a root node of the tree.*

A node $v \in V$ *such that* $lab(v) \in \Sigma$ *is called* element node *or simply* element*. A node* $v \in V$ *such that* $lab(v) \in \Gamma$ *is called* text node.

Definition 12. *A* regular tree grammar (RTG) *is a 4-tuple* $G = (N, T, S, P)$*, where:*

- N *is a finite set of non-terminals,*
- T *is a finite set of terminals,*
- S *is a set of start symbols, where* $S \subseteq N$*, and*
- P *is a finite set of productions of the form* $A \to aR$*, where* $A \in N$*,* $a \in T$*, and* R *is a regular expression over* N*.* A *is the* left-hand side*,* aR *is* the right-hand side*, and* R *is the* content model *of the production.*

Definition 13. *Given the alphabet* N*,* a regular expression *(RE) over* N *is inductively defined as follows:*

- $\emptyset$ (empty set) *and* ϵ (empty string) *are REs.*
- $\forall a \in N : a$ *is a RE.*
- *If* r *and* s *are REs over* N*, then* (rs) (concatenation)*,* $(r|s)$ (alternation) *and* (r^*) Kleene closure) *are REs.*

Note that DTD adds two abbreviations: $(s|\epsilon) = (s?)$ and $(ss^*) = (s^+)$. Also the concatenation is expressed using the '`,`' operator. The XML Schema language adds (among other extensions) another one, so-called *unordered sequence* of REs $s_1, s_2, ..., s_k$, i.e., an alternation of all possible ordered sequences of $s_1, s_2, ..., s_k$ (i.e., operator `all`, sometimes denoted as `&` as well).

Definition 14. *An* interpretation *χ of an XML tree $t = (V, E, \Sigma, \Gamma, lab, v_{root})$ against an RTG $G = (N, T, S, P)$ is a mapping from each node $v \in V$ to a non-terminal from N, denoted $\chi(v)$, such that:*

- *$\chi(v_{root}) \in S$, and*
- *for each node $v \in V$ and its subordinates v_0 , v_1 , ..., v_i, there exists a production $A \to aR \in P$ such that*
  - *$\chi(v) = A$,*
  - *$lab(v) = a$, and*
  - *$\chi(v_0)\chi(v_1)...\chi(v_i)$ matches R.*

Note that we interpret every text value by the value $pcdata$ for convenience.

Definition 15. *A tree t is* valid against *or* generated by *an RTG G if there exists an interpretation χ of t against G. A* regular tree language *is the set of trees generated by an RTG.*

A language generated by a grammar can be accepted by an automaton, in our case a finite state automaton.

Definition 16. *A* deterministic finite state automaton (DFSA) *is quintuple*

$$A = (Q, \Sigma, \delta, s, F),$$

where:

- *Q is a finite set of* states*,*
- *Σ is a finite set of* input symbols *(alphabet),*
- *$\delta : Q \times \Sigma^* \to Q$ is the* transition function*,*
- *$s \in Q$ is the* initial state*, and*
- *$F \subseteq Q$ is the set of* final states.

The language recognized by an automaton A is denoted $L(A)$.

In some approaches the definition of deterministic finite state automaton is extended to a *deterministic probabilistic finite state automaton*.

Definition 17. *A* deterministic probabilistic finite state automaton (DPFSA) *is a tuple*

$$A = (Q, \Sigma, \delta, q_0, \lambda, F, P),$$

where:

- *Q is a finite set of states, such that $\lambda \notin Q$,*
- *λ is a dummy state indicating immediate halt,*
- *Σ is an alphabet (finite set of symbols),*

- $\delta : (Q \times \Sigma) \to (Q \cup \{\lambda\})$ *is a transition function,*
- $q_0 \in Q$ *is an initial state,*
- $P : \delta \to \mathbb{N}_{\not{0}}$ *is function returning* transition use counts,
- $F : Q \to \mathbb{N}_{\not{0}}$ *is function returning* final state counts.

Definition 18. *Let $A = (Q, \Sigma, \delta, q_0, \lambda, F, P)$ be a DFSA. Let (V, E) be a directed* underlying graph *of A, where $V = Q$ is a set of nodes and $E \subseteq Q \times Q$ is a set of edges defined as follows:*

$$(q_1, q_2) \in E \quad \textit{iff} \quad \exists a \in \Sigma : \delta(q_1, a) = q_2.$$

A probabilistic prefix tree automaton (PPTA) *is a DFSA whose underlying graph is a tree rooted at state q_0.*

Note that for each RE we can construct a DFSA and vice versa. Each automaton $A = (Q, \Sigma, \delta, s, F)$ can be viewed as a directed labeled graph $G_A = (V_A, E_A, lab_A)$ where $V_A = Q$ and $e = (e_x, e_y) \in E_A$ if $\exists s \in \Sigma^*$ such that $\delta(e_x, s) = e_y$. Then $lab(e) = s$.

Now, we can define classes of grammars that correspond to particular XML schema languages. In particular, we borrow from [81] definitions of *local-tree grammars* that correspond to DTD and *single-type tree grammars* that correspond to XML Schema. Note that RELAX NG corresponds to general regular tree grammars (Definition 12). Firstly, we define necessary competition of non-terminals.

Definition 19. *Let us have an RTG $G = (N, T, S, P)$. Two non-terminals $A, B \in N$ are* competing *with each other if there exist two productions $A \to aR_1$ and $B \to aR_2$, where $a \in T$ and R_1, R_2 are REs over N.*

Definition 20. *A* local tree grammar (LTG) *is a RTG without competing non-terminals. A tree language is a* local tree language *if it is generated by a local tree grammar.*

Definition 21. *A* single-type tree grammar (STTG) *is a RTG such that*

- *for each production, non-terminals in its content model do not compete with each other, and*
- *start symbols do not compete with each other.*

A tree language is a single type tree language *if it is generated by a single type tree grammar.*

The studied problem of XML schema inference can be described as follows: Being given an input set of XML trees $I = \{t_1, t_2, ..., t_n\}$, we search for an XML schema, i.e., an RTG $G_I = (N_I, T_I, P_I, S_I)$, such that $\forall i \in [1, n] : t_i$ is valid against G_I. In particular, we are searching for G_I that is enough concise, precise and, at the same time, general. This requirement indicates, that the optimal result is hard to define and, in general, there may exist several solutions, i.e., a set of candidate RTGs $O = \{G_I^1, G_I^2, ...G_I^m\}$, such that $\forall i \in [1, n], j \in [1, m] : t_i$ is valid against G_I^j, whereas we are looking for the optimal one $G_I^{opt} \in O$.

The problem of finding G_I^{opt} can be viewed as a special kind of optimization problem [153].

Definition 22. *A model $M = (\Theta, \Omega, \sigma)$ of a* combinatorial optimization problem (COP) *consists of a search space Θ of possible solutions to the problem (so-called* feasible region*), a set Ω of constraints over the solutions and an* objective function $\sigma : \Theta \to \mathbb{R}_0^+$ *to be minimized.*

In our case $\Theta = O$. As it is obvious from Definition 13, Θ is theoretically infinite and thus, in fact, we can search only for a reasonable suboptimum. Ω is given by the features of XML schema language we are focussing on, i.e., RTG, LTG or STTG (as described in Definitions 12, 20 and 21). And finally, to define σ we need to find a criterion that describes the quality of the given RTG, such as, e.g., the MDL principle [48], user evaluation etc., as we discuss later.

Most of the existing works use the same strategy consisting of the following phases:

- Phase I. Derivation of *initial grammar* (IG) describing the input data
- Phase II. Clustering of productions of IG
- Phase III. Inference of a RE for each cluster of IG, i.e., building the *output grammar* (OG)
- Phase IV. Refinement of the inferred REs in OG
- Phase V. Inference of simple data types, i.e., extension of OG
- Phase VI. Inference of integrity constraints, i.e., extension of OG
- Phase VII. Expressing the OG in the target XML schema language

The current approaches usually use simple clustering algorithms in phase II. And they also omit phase IV., V. and VI., i.e., refining and extending of the OG. The idea of *jInfer* is based on the observation, that the existing methods naturally focus on optimization of selected phases of the inference process. It models the phases of the inference process as modules, whereas the user is enabled to implement new ones, replace the existing ones and compare the results. In the following text we describe how this idea is implemented in *jInfer* and later we will introduce several improvements of the inference process implemented within *jInfer*.

6.5 Architecture and Implementation

The description of *jInfer* [66] architecture is divided into three main views: the data view, the process view and the platform view. The data view will commence with describing the data structures, namely representations of REs and XML elements, attributes, and simple data. The process view represents the inference process itself. Finally, we will look at *jInfer* as at a NetBeans Platform [154] application consisting of a number of modules performing their tasks and communicating together through a set of well-defined interfaces.

6.5.1 Data View

The basic data structure we need to represent is a RE. RE is implemented as class `Regexp<T>` with supporting classes `RegexpInterval` and `RegexpType`. Each `Regexp<T>` instance has a type (property `type`) of `RegexpType` enumeration:

- Lambda (λ) – empty string,
- Token – a letter of the alphabet,
- Concatenation – one or more REs in an ordered sequence, e.g., (a, b, c, d),
- Alternation – a choice between one or more REs, e.g., $(a|b|c|d)$, or
- Permutation – shortcut for all possible permutations of REs (our notation is $(a\&b\&c\&d)$.

Using Java generics, `Regexp<T>` can represent RE over any alphabet. RE is in fact an n-ary tree. For example expression $(a, b, ((c|d), e), f)$ can be viewed as a tree depicted in Figure 6.5. We implement this tree using a property of `Regexp<T>` class called `children`, which is of type `List<Regexp<T>>`.

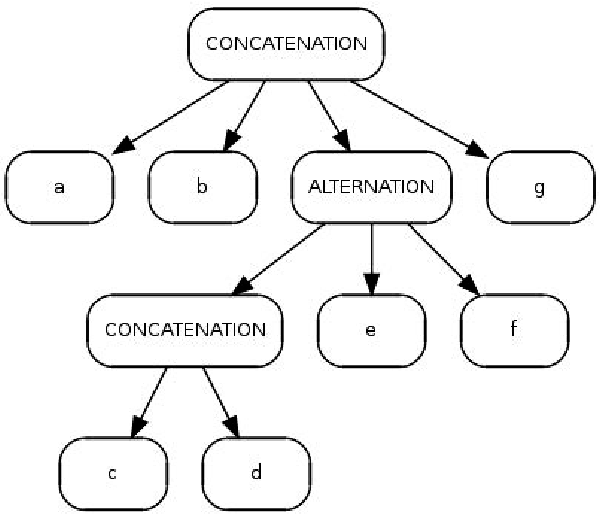


Figure 6.5: A sample tree for RE $(a, b, ((c|d), e), f)$

The second most important data structures are the XML data. XML data basically encompasses elements, text nodes and attributes. For maximum generality, we break apart these objects and we define three basic interfaces:

- `NamedNode`,
- `StructuralNode`, and
- `ContentNode`.

The first one stands for a bare node in XML document tree, it has its name and *context* within the tree (a path from root node). The latter two extend `NamedNode` interface. `StructuralNode` is for nodes which form structure of XML document tree: elements and text nodes. `ContentNode` is for nodes that have content in XML documents: text nodes and attributes. We have three classes: `Element` for elements, `SimpleData` for text nodes, `Attribute` for attributes. For even more generality in design, we decided to implement two abstract classes at the mid-level:

- `AbstractNamedNode` implements methods from `NamedNode` interface to handle context, name and metadata.
- `AbstractStructuralNode` implements only the task of deciding if the instance is `Element` or `SimpleData`.

The interface/class model for representing XML data is depicted in Figure 6.6.

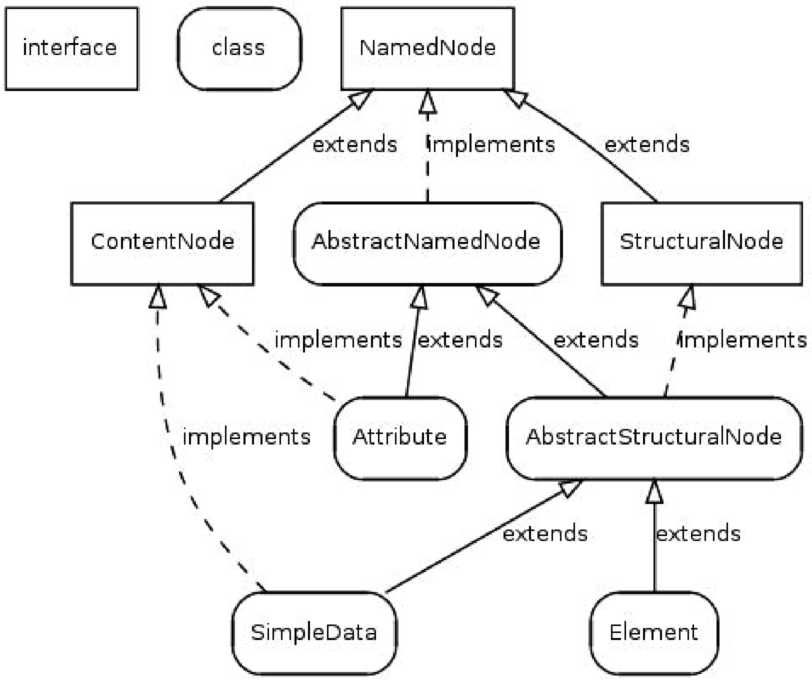


Figure 6.6: XML representation in *jInfer*

6.5.2 Process View

The inference process in *jInfer* corresponds to the before specified phases; however, as depicted in Figure 6.7, some of the phases are coupled into one module. Nevertheless, the modules have further submodules as we will see later.

From the high-level viewpoint, the inference process in *jInfer* consists of three consecutive steps carried out by three different modules:

1. *IG generation*, covering phase I. of the inference process, is done by the `Initial Grammar Generator (IGG)` module. This is the process of converting all of the inputs to IG representation. All documents, schemas and queries selected as input are evaluated, simple productions are extracted and sent to the next step. For example, the following XML document fragment:

```
<person>
  <info>
```

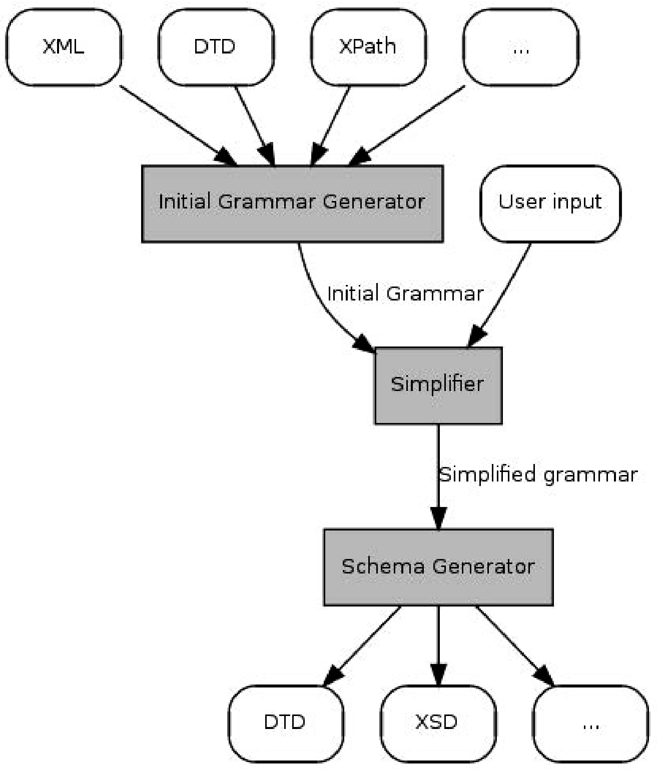


Figure 6.7: High-level view of the inference process in *jInfer*

```
    Some text
    <note/>
  </info>
  <more/><more/><more/>
</person>
<more/>
<person>
  <more/>
</person>
```

is translated into the following IG productions:

$$\begin{array}{rcl}
person & \rightarrow & info, more, more, more \\
info & \rightarrow & simple\_data, note \\
note & \rightarrow & empty\_concatenation \\
more & \rightarrow & empty\_concatenation \\
person & \rightarrow & more, more, more \\
more & \rightarrow & empty\_concatenation \\
more & \rightarrow & empty\_concatenation \\
more & \rightarrow & empty\_concatenation
\end{array}$$

Note that in this case the process is very simple. However, complications can bring new input data (as we will see later), such an obsolete schema, queries etc. Then this module has to process more types of input and prepare respective productions.

2. *Simplification*, covering phases II. – VI. of the inference process (in some cases using empty modules), is done by the `Simplifier` module. This is the process of simplifying the IG, i.e., describing it using a smaller number of (more complex and more general) productions. User interaction might be used in this step to help to achieve better simplification – for instance the user can specify which states should be merged (see Figure 6.8 representing a screenshot of *jInfer*). For example, the previous productions (clustered according to element name) could be simplified to a single production for element `person`:

   $$person \rightarrow info?, more\{1,3\}$$

   Productions for elements `info`, `more` and `note` after simplification will be:

   $$\begin{aligned} info &\rightarrow simple\_data, note \\ note &\rightarrow \lambda \\ more &\rightarrow \lambda \end{aligned}$$

   This module represents the key part of the inference process. As we will show later, it consist of other submodules corresponding to particular phases or their parts.

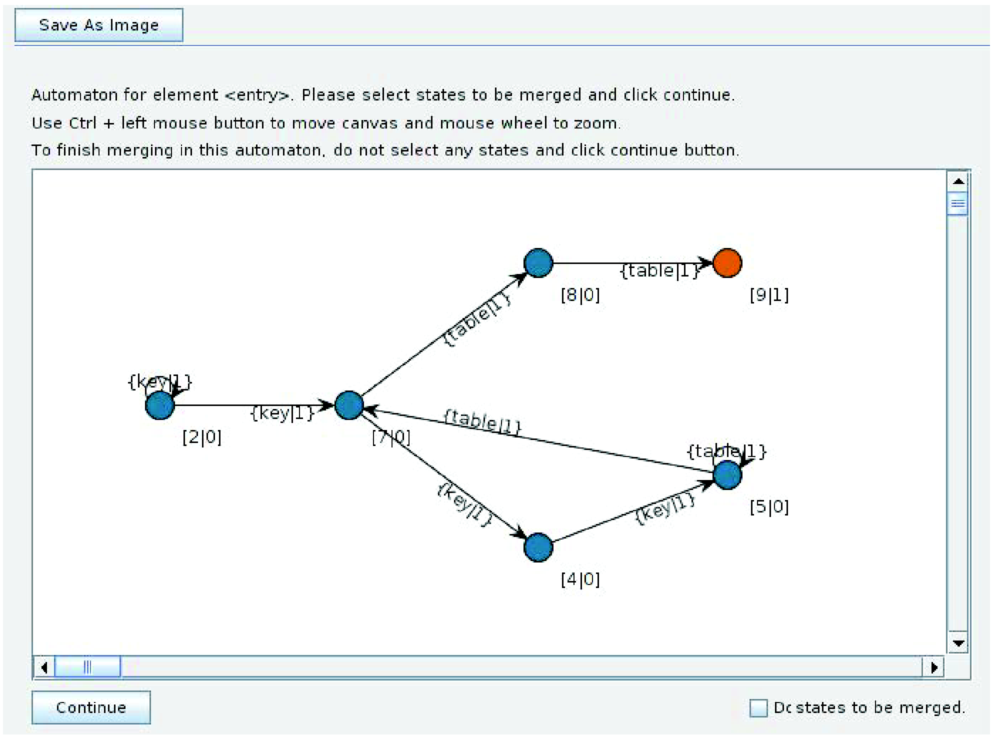
3. *Schema export*, covering phase VII. of the inference process, is performed by the `Schema Generator (SchemaGen)` module. This is the process of creating the resulting schema file from the simplified productions. The result of this step is a textual representation of the schema, which is sent back to the framework (and later displayed, saved etc). For the previous simplified productions, the resulting DTD would be:

   ```
   <!ELEMENT person
             (info?, more, more?, more?)>
   <!ELEMENT info (#PCDATA | note)*>
   <!ELEMENT note EMPTY>
   <!ELEMENT more EMPTY>
   ```

   Note that the process is not so straightforward. For instance, for element `person`, when the simplified grammar specifies its occurrence to at least once and at most 3 times, as DTD has no such construct, the export module has to find a suitable expression. The situation is even worse for elements that contain simple data within a simplified production. The only way to express mixed content in DTD is to use `(#PCDATA | note)*` expression. Even if the RE is complicated, the export module has to do this “flattening”.

6.5.3 Platform View

Being a NetBeans Platform application, *jInfer*, is divided into multiple *NetBeans* modules [155]. Theoretically, the whole *jInfer* could be contained in a single NetBeans



Automaton for element <entry>. Please select states to be merged and click continue.
Use Ctrl + left mouse button to move canvas and mouse wheel to zoom.
To finish merging in this automaton, do not select any states and click continue button.

Figure 6.8: *jInfer* interface for merging states of automaton

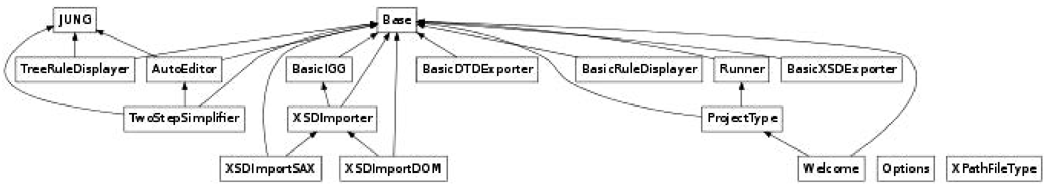


Figure 6.9: *jInfer* module dependencies

module, because the framework uses *lookups*[2] to locate logical modules. However, for the sake of organization, *jInfer* is split into several modules and most of the time there is a strong correlation between logical units of *jInfer* and its NetBeans modules.

A short overview of these modules and their functions follows. The dependencies between all the modules are depicted in Figure 6.9. To get a deeper understanding of any of them, please, refer to the respective documentation [156].

The *basic* modules are as follows:

- `jInfer`: It is not a NetBeans module in fact, but rather a module suite – the "root" NetBeans project for the whole framework.

- `Base`: It contains the common data structures, interfaces, and utility logic shared across all other modules.

The *inference* modules are as follows:

- `Runner`: This module encapsulates the three-step inference process described in Section 6.5.2.

[2]A part of the NetBeans API which provides a way how to achieve late-binding between code based on this feature.

- `BasicIGG`: This module contains an extensible implementation of `IGGenerator`. The basic version handles XML documents, DTDs and XPath queries.
- `XSDImport`: Demonstrating extensibility of `BasicIGG`, this is a basic module for XSD input.
- `XSDImportSAX`: XSD input handling using SAX parser [111].
- `XSDImportDOM`: XSD input handling using DOM parser [157].
- `TwoStepSimplifier`: An implementation of `Simplifier`. This particular module implements merging state algorithm from [47] with basic handling of attributes – see Section 6.5.4.
- `BasicDTDExporter`: An implementation of `SchemaGen` with output to DTD.
- `BasicXSDExporter`: An implementation of `SchemaGen` with output to XSD.

The *utility* modules are as follows:

- `BasicRuleDisplayer`: This module is capable of displaying a grammar in a graphical way.
- `TreeRuleDisplayer`: This is more advanced version of the former using `JUNG`'s graphing capabilities [158].
- `JUNG`: This is a wrapper module for the JUNG.
- `AutoEditor`: This is an encapsulation of an interactive finite state automaton editor (see Figure 6.8).
- `Options`: A NetBeans specific module which adds a *jInfer* category into NetBeans *Options* window.
- `ProjectType`: A NetBeans specific module which defines a *jInfer* project type, along with its input and output files, settings etc.
- `XPathFileType`: An NetBeans specific module which enables NetBeans to recognize a `.xpath` file extension.

Last but not least, Figure 6.10 shows default implementation of three modules representing the three steps of the inference process.

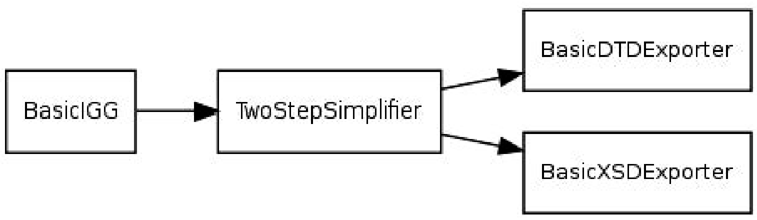


Figure 6.10: Default inference modules in *jInfer*

6.5.4 Default Simplifier `TwoStepSimplifier`

As mentioned before, `TwoStepSimplifier` is a sample implementation of `Simplifier` provided in *jInfer* as a default. Like any of the *jInfer* modules, `TwoStepSimplifier` can be replaced with a user-provided module. In addition, its inner strong modularity ensures that it can be extended easily, i.e., only a part of its implementation can be modified/extended.

Modular Design

`TwoStepSimplifier` is inspired by the design described in [82]. The inference proceeds in two steps:

1. Clustering element instances into clusters of (probably) the same elements.
2. Inferring the RE from examples of element contents taken from all elements in a cluster.

Clustering is delegated to `Clusterer` submodule, and the task of RE inference for each cluster is delegated to `ClusterProcessor` submodule. There is also a third submodule called `RegularExpressionCleaner`. Its purpose is to refine the output REs. The hierarchy of modules is depicted in Figure 6.11.

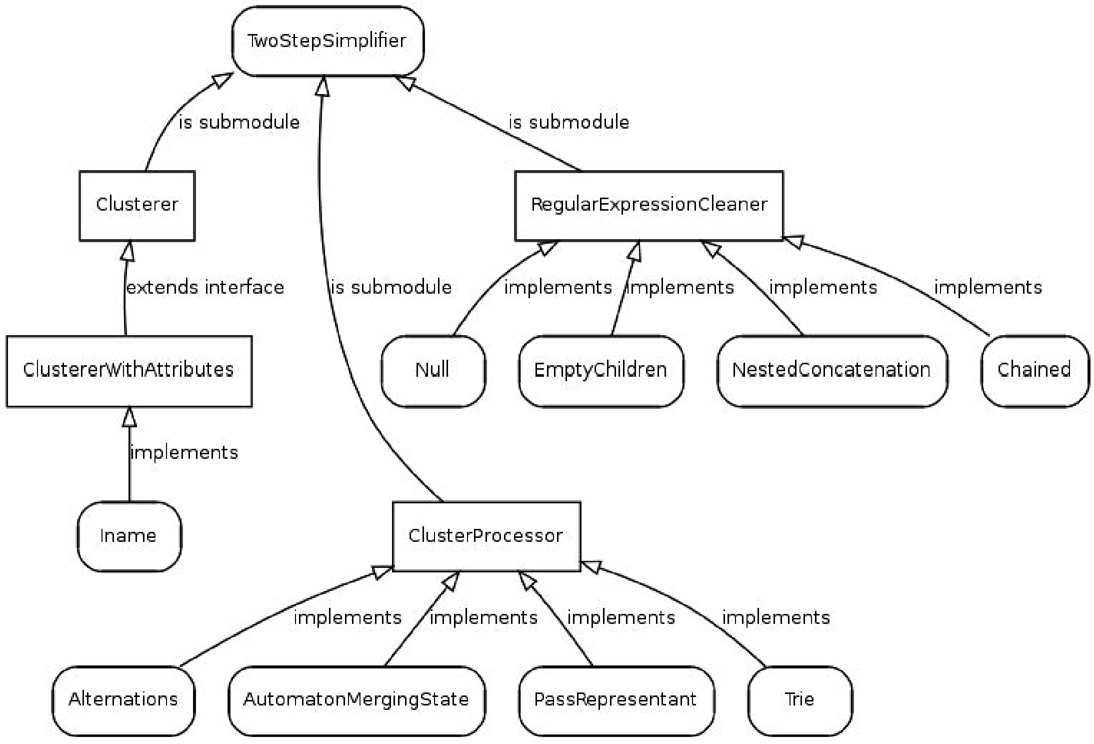


Figure 6.11: Submodules of `TwoStepSimplifier` with their implementations

As we can see, we provide one `Clusterer` (or `ClustererWithAttributes` which is its specialization that enables clustering with additional input information), four `ClusterProcessor` implementations and four `RegularExpressionCleaner` implementations (including a `null` one which just returns the given RE). The `ClusterProcessor` implementations are as follows:

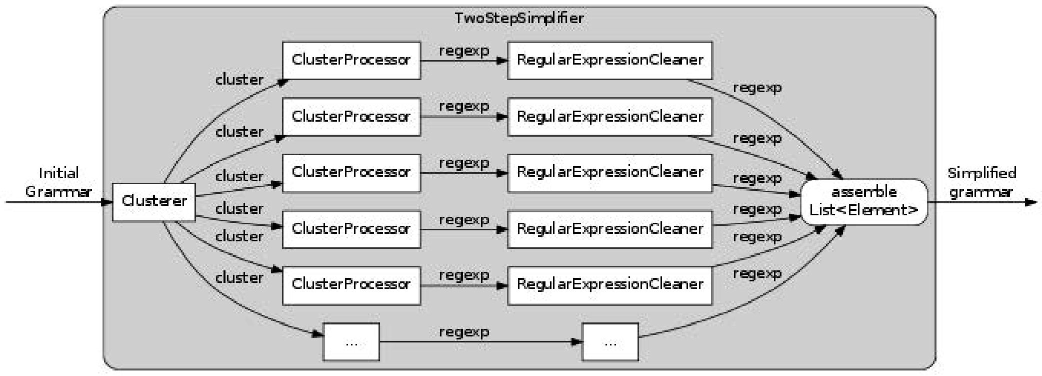


Figure 6.12: Data flow in `TwoStepSimplifier`

- `Alternations` processor simply gets all right-hand sides od productions in the cluster and creates their alternation as the output RE. No generalization or simplification is done.
- `AutomatonMergingState` is an implementation of a classical merging state algorithm from [47].
- `Trie` takes all productions in a cluster, treats them like strings and builds a prefix tree (a "trie") of them. The tree is the final RE.
- `PassRepresentant` processor returns for each cluster its representative general production. This has nothing to do with grammar simplification, it is just a proof-of-concept submodule.

The `RegularExpressionCleaner` implementations are as follows:

- `EmptyChildren` cleaner wipes out REs of type concatenation, alternation, permutation, which have empty `children` member, such as, e.g.,

$$(name, (), (person, id)).$$

- `NestedConcatenation` cleaner wipes out unnecessary nesting, such as, e.g.,
  $(name, (person, id))$.
- `Chained` cleaner allows the user to chain more cleaners with output from the first one plugged as input into second one and so on.

Data Flow

The data flow in `TwoStepSimplifier`, depicted in Figure 6.12, can be summarized as follows:

1. The input IG is sent to `Clusterer` to be clustered.
2. Each cluster is sent to submodule `ClusterProcessor` which returns a RE for that cluster.

3. The RE is sent to `RegularExpressionCleaner` submodule for cleaning.
4. All REs representing all the processed clusters are added to the list of simplified grammar productions.
5. The list of simplified productions is returned (as the OG).

In the following sections we will show how the default implementation, i.e., an existing acknowledged solutions to the problem of XML schema inference, can be further extended. In this area we have already proposed several approaches. However, we will show three demonstrative examples:

1. General optimization of the inference process (Section 6.6),
2. Exploitation of a new type of input – XML queries (Section 6.7), and
3. Inference of a new type of output – Schematron schema (Section 6.8).

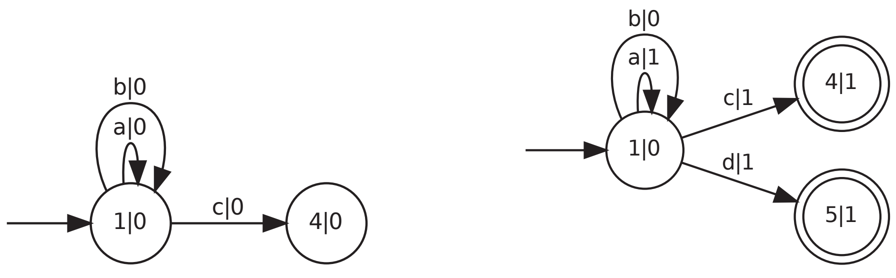
6.6 Inference Optimization

In this section we focus on optimization of the first three phases of the inference process. In particular, we will describe the following extensions:

- exploitation of a (possibly existing) obsolete XML schema in phase I.,
- a new (more accurate) measure of quality of intermediate result automaton in phase III., and
- a possibility to automatically tag the selected input grammar productions as invalid in phase II., excluding them from inference (e.g., deviations, misspelled words, etc.) ensuring generating of a more accurate schema for valid inputs.

The extensions were first proposed and are described in detail in [159]. Here we provide their overview in the context of *jInfer* to demonstrate its universality and modularity.

We follow the inference steps proposed in [82]. In phase II., *positive examples* (element instances from input documents) are clustered according to not only element names, but also similarity of their context and content, grouping the instances corresponding to one element type definition into one cluster. Contrary to existing works, we modify phase I. and we also parse XML schema files (i.e., the obsolete ones) into grammar productions. These are then clustered according to the element name together with element instances originating from XML documents. Each cluster hence contains at most one production from XML schema input files and zero or more productions from XML documents, all forming the IG. Since in both XSD and DTD the element content model is basically specified by a RE, we consider positive examples as being generated by some DPFSA and try to infer this automaton. If there is a RE from schema input, then a *DPFSA torso* is constructed from it (see Figure 6.13 (a)). Starting with an empty automaton or with a torso (when an obsolete schema is processed), we run the same algorithm for building DPFSA in the form of PPTA from positive examples and merge all the productions (see Figure 6.13 (b)). (The numbers after the '|' character represents the transition use counts or the final state counts.)



(a) DPFSA torso from RE $(a|b)^*c$ (b) After parsing inputs ac and d

Figure 6.13: An example of building DPFSA as PPTA starting from DPFSA torso

In phase III., the automaton is modified by merging its states – in this case we apply several improvements, as described later. The inferred automaton is then converted into an equivalent RE using a *state removal algorithm* [49] and the RE is added to OG.

In other words, similarly to the existing papers, we omit phases IV. – VI. and we do not modify phase VII. However, thanks to modularity of *jInfer*, in these cases we can re-use the existing modules. In addition, since *jInfer* provides all the necessary "side" aspects of schema inference, such as parsing of XML data and checking their well-formedness or visualization of input data, automata, productions,etc., we can focus purely on the extensions themselves.

6.6.1 Selecting States to Merge

To select states to merge in a reasonable way we employ two verified state equivalence criteria: sk*-strings* [46] heuristic criterion and k, h*-context* [47] criterion. Modules responsible for providing candidate alternatives for merging are called *merge criterion testers*. From the available alternatives a *merging state strategy* selects which to merge and which not. In *jInfer* we have implemented several merging state strategies called: *Greedy*, *GreedyMDL*, *HeuristicMDL* and *DefectiveMDL*. The first three are classical approaches provided with *jInfer* as a default, the last one is our own new proposal (see Section 6.6.3).

Greedy The *Greedy* strategy simply merges all candidate states provided by merge criterion testers. For example for the k, h*-context* tester it means, that it simply creates a k, h*-context* automaton as defined in [47].

GreedyMDL The *GreedyMDL* strategy uses the MDL principle to evaluate a DPF-SA and input strings encoded by the automaton. The precise MDL code (called an *objective quality function*) is described in Section 6.6.2. For now, it is sufficient to assume the existence of an objective function $mdl(A, S)$ which is given an automaton A and a set of input strings S and returns a non-negative real value, the overall quality of the solution, where a lower value signifies a better solution, i.e., the quality the automaton A describes strings in S.

While trying to merge candidate alternatives, the *GreedyMDL* strategy always keeps the currently achieved minimum quality value (and the associated automaton). A space of possible solutions is explored in a greedy way, but some sort of a complete scanning of continuation possibilities is done: all candidate alternatives to merge are evaluated. The algorithm stops when there are no more candidates to merge, or when all alternative candidates returned by merge criterion testers end up in an automaton with higher quality value than the one actually achieved.

HeuristicMDL The *HeuristicMDL* as a simple heuristic strategy works basically the same way as *GreedyMDL*, but it holds n best minimal solutions instead of only one. At each iteration, merge criterion testing for one randomly selected automaton of the n best automata is done. All the alternatives returned are attempted to be merged and only the automaton with quality value lower than the current worst solution is stored in capacity-constrained sorted list (thus, it always holds the best n solutions). The algorithm stops when it is "staggering" – when the set of the best n solutions is not modified for a whole iteration.

6.6.2 Objective Quality Function

In the sense of the crude MDL, one has to design a code for a hypothesis and a code for data compressed using the hypothesis. Since in this work a basic assumption is that positive examples were generated by some DPFSA, the hypothesis is the DPFSA itself. And as described in [160, p. 100], if a hypothesis is of probabilistic character, the best code to use is the complete prefix code with code-lengths equal to $-\log(p)$ for the one option, whose probability of appearance in data equals to p. When generating strings using the DPFSA $A = (Q, \Sigma, \delta, q_0, \lambda, F, P)$, in each state of the automaton the algorithm decides which transition to follow or whether to output a whole word randomly – driven by a *probabilistic density function* defined by the probabilities of each followed transition, and the actual state that it is final. Given a state q, we compute a unity value:

$$u_q = F(q) + \sum_{q' \in Q, a \in \Sigma} P(q, a, q') \tag{6.1}$$

Then, function $P'_q : \Sigma \times (Q \cup \{\lambda\}) \to [0, 1]$ is defined as:

$$\begin{aligned} P'_q(a, q') &= \tfrac{P(q,a,q')}{u_q} && (\forall q' \in Q, a \in \Sigma) \\ &= 0 && (q' = \lambda) \end{aligned} \tag{6.2}$$

Function P'_q together with the value $f'_q = \frac{F(q)}{u_q}$ forms a probabilistic density function of a discrete probability random variable X_q i.e., "what is done next, if we are in state q" defined as:

$$P[X_q = (a, q')] = P'_q(a, q') \tag{6.3}$$

$$P[X_q = terminate] = f'_q \tag{6.4}$$

Using the set of random variables X_q (one for each state q), encoding input strings is simple: When the automaton generates a string, the configuration sequence is the same as if it had the input string which the automaton was accepting. Thus, computing a code-length can be done as follows: For each input string, traverse automaton while reading it and record probabilities of transitions along the way. Let us consider input string $s = a_1, \ldots, a_n$. Let probabilities $p_1, \ldots, p_n$ be recorded transition probabilities, and p_{n+1} the probability f'_q of the state, where reading of the string ended. Code-length C of string s equals to:

$$C(s) = \sum_{i=1}^{n+1} -\log(p_i) = -\log\left(\prod_{i=1}^{n+1} p_i\right) \tag{6.5}$$

However, there is no need to traverse the automaton with each input string to get the total code-length of all input strings. When the DPFSA is built, each input string incremented the use count value of each transition passed and incremented the final count value of the state it ended in. When a DPFSA is traversed for each input string, each transition is passed exactly its use count-times and traversing ends in each state exactly its final count-times. From this, it is easier to compute the total code-length of input strings S (encoded with the help of DPFSA A, where $\{u_q | q \in Q\}$ are pre-computed unity values for each state) as:

$$L(S|A) = \sum_{q \in Q, a \in \Sigma, q' \in Q} \left(\begin{array}{c} P(q,a,q') \\ \times \\ -\log\left(\frac{P(q,a,q')}{u_q}\right) \end{array} \right) \tag{6.6}$$

The key problem is how to encode the DPFSA. In general, there is no universal way, because DPFSA is an ad-hoc model to solve a custom ad-hoc problem and we propose an ad-hoc code for it. Let us denote the states of an automaton as $q_1, \ldots, q_{|Q|}$. The proposed code is $< |Q|, |\Sigma|, alphabet, \langle q_1 \rangle, \ldots, \langle q_{|Q|} \rangle >$, where:

- $|Q|$ is the cardinality of the set of states encoded using *standard universal code for integers* (SUCI), and
- $|\Sigma|$ is the cardinality of Σ encoded using SUCI.

The symbols of an alphabet are named using a uniform code in the table $alphabet$, which is a translation table from uniform encoding of each symbol into a prefix code. The prefix code for alphabet symbols is established using a histogram of symbol occurrences over all transitions of the automaton. The probability of an individual symbol a for this code is therefore:

$$p_a = \frac{\text{occurences of symbol } a}{\text{occurences of all symbols}} \tag{6.7}$$

Each $\langle q_i \rangle$ is a code of one state in the automaton $\langle |Q'_i|, \langle t_1 \rangle, \ldots, \langle t_{|Q'_i|} \rangle \rangle$, where Q'_i is a set of all states immediately reachable from the state q_i, formally $Q'_i = \{q; q \in Q, \exists a \in \Sigma : \delta(q_i, a) = q\}$. Each $\langle t_j \rangle$ is a code of one out-transition of the state q_i, each in a form $\langle q, P(q,a), a \rangle$, where q is a code for a destination state encoded using a uniform code over all states, $P(q,a)$ is a use count value of the transition encoded using SUCI, a is a symbol of alphabet encoded using the prefix code for the alphabet established earlier. Let us denote p_a the probability of each symbol a in the alphabet. Then the code-length of this ad-hoc code of the automaton is:

$$\begin{aligned} L(A) \quad &= suci(|Q|) + suci(|\Sigma|) + |\Sigma| \cdot \log(|\Sigma|) - \\ &\log\left(\prod_{a \in \Sigma} p_a\right) + \left(\sum_{i=1}^{|Q|} suci(|Q'_i|)\right) + \\ &\sum_{i=1}^{|Q|} \sum_{j=0}^{|Q'_i|} \left(\log(|Q|) + suci(P(q,a)) - \log(p_a)\right) \end{aligned} \tag{6.8}$$

where $suci(|Q|)$ is the SUCI length for state-count, $suci(|\Sigma|)$ is the SUCI length for alphabet size, $|\Sigma| \cdot \log(|\Sigma|)$ is the sum of lengths of each left side in the *alphabet translation table* (each symbol with code-length $\log|\Sigma|$), $-\log\left(\prod_{a\in\Sigma} p_a\right)$ is the sum of lengths of each right side in the alphabet translation table (each symbol a with code-length $-\log(p_a)$), $\left(\sum_{i=1}^{|Q|} suci(|Q'_i|)\right)$ is the sum of all SUCI lengths for transition counts of each state, and the last double sum is the sum of the sum of code-length of each transition, destination state with uniform code-length of $\log|Q|$, SUCI length for use count and prefix code-length for a symbol a from the alphabet. The whole code-length function is given as the sum of Equations (6.6) and (6.8):

$$mdl(A,S) = L(S|A) + L(A) \tag{6.9}$$

6.6.3 *DefectiveMDL* Merging State Strategy

In general, using *jInfer* the user can chain merging state strategies arbitrarily. The *DefectiveMDL* strategy should be attached at the end of such a chain, when the automaton is ready to be converted into a RE. It is based on the idea to decide which input strings are so eccentric that they probably are "mistakes" and should be repaired in input documents rather than incorporated into the output schema. In some cases, input documents are selected to cover all expected constructs and, thus, there is no use for DefectiveMDL strategy, but in case of automatic inference based on "dirty" documents, the identification of "mistakes" can be useful.

Let T be the set of input strings we suspect as eccentric. We try to remove T from the inference process. If the inferred schema is much simpler, we consider T as eccentric. Apparently, this approach would simply remove all input strings, since no documents fit the simple `EMPTY` construct. Here, we exploit MDL again and we try to remove input strings from set T. If $mdl(A, S \setminus T)$ is smaller enough than $mdl(A,S)$, we consider T as eccentric.

To formalize the "smaller enough" metric, we define a criterion: When the description length of a new automaton and input strings, together with the description length of the removed input strings, is smaller than the description length of an old automaton together with all input strings, the strings are considered eccentric. We define an *error code* for one input string $a_1, \ldots, a_n$ as a sequence of prefix codes for each symbol (specified the same way as in Equation (6.8)). Thus, the code-length of the error code for one input string equals to:

$$L_{error}(s) = \sum_{i=1}^{n} -\log(p_{a_i}) = -\log\left(\prod_{i=1}^{n} p_{a_i}\right) \tag{6.10}$$

where p_{a_i} is the probability of symbol a_i in the established prefix code for the alphabet. Since by removing input strings it may occur that we also remove some symbol from the alphabet used in the automaton, the prefix code for the alphabet is established with a histogram not only over the automaton, but also over removed input strings. Basically, the prefix code for the alphabet remains the same, since it has to encode all input strings, no matter if they are used in the automaton or in the error code. So the code-length of the error code of all removed strings is:

$$L_{error}(S) = \sum_{s\in S} L_{error}(s) \tag{6.11}$$

We denote $mdl(A, S, S_r)$ the MDL code-length of an automaton A, strings S encoded using the automaton A, and strings S_r removed. Then $mdl(A, S, S_r)$ equals to:

$$mdl(A, S, S_r) = mdl(A, S) + L_{error}(S_r) \tag{6.12}$$

The idea of MDL comparison can be likened to a compression of a text document using zip compression: When the length of the zip-file plus the length of some removed sentences from the document is smaller than the length of the original zip-file with all sentences, it makes sense to deduce from the phenomena that the removed sentences are so eccentric that they corrupt underlying data regularity.

The input strings can be removed from the automaton as follows: The automaton is traversed while reading an input string (remember that the automaton is deterministic) and each transition gets its use count value decremented along the way. The final state gets its final count value decremented. Since only strings that previously formed the automaton are removed, use counts and final counts never reach negative values.

The last question is which input strings to remove. In *jInfer*, we use a program interface called *Suspect*, which returns input strings it is suspecting as eccentric. Checking strings one by one is one simple strategy implemented. If we remove all input strings that pass one transition, the transition is rendered as unused.

6.6.4 Conclusion

As we can see, the universality and modularity of *jInfer* enables one to consider any kind of extension of the classical inference processes used in the current literature. The author is just expected to determine the submodules (s)he wants to modify or completely replace. All other parts of the inference process can be re-used without any necessary intervention. From the opposite point of view, when the *jInfer* tool is, e.g., extended with a new, more user-friendly way of visualization of automata, all the implemented inference modules remain the same and just exploit the new capability.

6.7 Schema Inference with the Knowledge of XML Queries

In this section we describe an approach which extends the inference process with additional input information, namely XML queries, and focuses on the often omitted phases. We will describe the following extensions:

- adding phase V. (inference of simple data types),
- adding phase VI. (inference of ID/IDREF(S)).

In both the cases we exploit the input XML queries expressed in XQuery [21] and the proposal is intended to be implemented as an extension of an existing inference method processing XML documents. Hence, we result from the default inference method of *jInfer*, i.e., `TwoStepSimplifier`, and we add new modules representing the two phases. In [161] we provide the full description of the approach, including introductory discussion of many other possible ways of exploitation of information from XQuery queries in the inference process. From them we select only several most interesting representatives and we deal with them in this section.

The proposed extension consists of the following four main steps:

1. **Construction of a syntax tree of each XML query**. We use lexical and syntax analyses proposed in [162] and for each XQuery on input, we construct a data structure called a *syntax tree*.

2. **Static analysis of expression types**. The algorithm searches for particular expressions in the syntax trees and statically (without evaluation) determines their types.

3. **Inference of simple data types**. When the target types of expressions are determined, they are utilized to infer simple data types of elements and attributes.

4. **Key discovery**. The final step is an extension of approach [50] inferring keys and foreign keys.

6.7.1 Step 1: Construction of a Syntax Tree

The first step of the algorithm involves lexical and syntax analyses [162] known from the construction of compilers and produces a syntax tree. Since they are not directly related to inference, and, thus, they are not directly related to the topic of this paper, we will not describe them. Nevertheless, they provide us with a helpful processing of XQuery queries and we can focus on the inference process.

Definition 23. *A* syntax tree *of XQuery query Q is tuple $T = (V, E, c, \mathcal{P}, o)$ where*

- *$V \subset \mathbb{N}$ is a set of nodes, each node representing a particular XQuery construct in query Q,*
- *E is a set of pairs (v, w) where $v, w \in V$ and for every $a, b \in V, a \neq b : (a, b) \in E$ if and only if a construct represented by b is a direct component of a construct represented by a (b is a child of a) in query Q,*
- *$c : V \rightarrow C$ is function assigning each node with its class from set C of all XQuery language constructs (listed in [162], e.g.,* `LiteralNode`*,* `AttributeNode`*,* `Node`*,* `ExprNode` *etc.),*
- *$\mathcal{P}$ is a set of functions specifying additional properties of the nodes and distinguishing the nodes of the same classes, and*
- *$o : V \rightarrow \mathcal{O}$ is a function specifying an order of children of the nodes, where $\mathcal{O} = \{o_v : E_v \rightarrow \mathbb{N} | v \in V, E_v = \{(v, w) | w \in V, (v, w) \in E\}\}$ is set of functions specifying the children order for each node. For every $v \in V, o(v) = o_v$ so that $o_v(v, w)$ is a sequential number of a construct represented by w amongst constructs represented by children of v in query Q.*

Regarding the additional properties, two constructs in Q represented by two nodes of the same class from C may differ in certain ways, and, therefore, it is necessary to distinguish them. For instance, two different literal values in Q are represented by nodes $l_1, l_2 \in V$ and $c(l_1) = c(l_2) =$ `LiteralNode` but each has a different value and type. Therefore $\mathcal{P}$ contains functions:

$$type_{LiteralNode} : V_{LiteralNode} \rightarrow Types_{literal}$$
$$value_{LiteralNode} : V_{LiteralNode} \rightarrow Values_{literal}$$

where $V_{LiteralNode} = \{v | v \in V, c(v) = \texttt{LiteralNode}\}$, $Types_{literal} = \{\texttt{DECIMAL}, \texttt{INTEGER}, \texttt{DOUBLE}, \texttt{STRING}\}$, and $Values_{literal}$ is a set of all literal values (all valid XQuery decimal numbers, integers, double numbers and strings).

The set $\mathcal{P}$ contains other similar functions but due to their large number, we do not define them formally. Functions $varName_{VarRefNode}$, $axisKind_{AxisNode}$, or $operator_{OperatorNode}$ are examples of commonly used functions from $\mathcal{P}$. Their meaning will be explained in a place of their usage. For details, see [162].

The node classes of the syntax are organized in an *is-a* hierarchical structure, commonly used in the object oriented programming languages, where an object can be of several types. For example, an instance of the syntax tree cannot directly contain nodes of `Node` and `ExprNode` classes (for every $v \in V, c(v) \neq \texttt{Node}, c(v) \neq \texttt{ExprNode}$), but it can contain nodes of `AttributeNode` and `LiteralNode` classes. Regarding the multiplicity of types, a node of `LiteralNode` class is also considered to be of two indirect types: `ExprNode` and `Node`.

An important characteristic of the syntax tree is related to definition of local variables and their scope in the XQuery language. The representation of a definition of a local variable in the syntax tree is a node of `VariableBindingNode` class. Nodes of that class have only two children; a node representing the type of the variable and a node representing the binding expression (expression defining the value of the variable, and thus, it cannot use the variable). Hence, the entire subtree does not contain any expressions that use the variable. Therefore, the scope of the new variable is not the subtree of the `VariableBindingNode` class node. It depends on the type of XQuery construct of which the variable binding is an (indirect) component.

For example, a syntax tree constructed from the following query is shown in Figure 6.14:

```
declare namespace local =
                  "http://www.foobar.org";
declare function local:convert
    ($v as xs:decimal?)
  as xs:decimal? {
    2.20371 * $v
};

for $i in /site/open_auctions/open_auction
return
  local:convert(zero-or-one($i/reserve))
```

6.7.2 Step 2: Static Analysis of Expression Types

In the second step, we statically (i.e., without evaluation of the queries) determine types of expressions in the syntax tree. Information on the types of expressions can be used by consecutive steps of the algorithm. The consecutive steps described in this paper will not use the determined types of all expressions; however, this step may be useful in a future extending.

The analysis of expression types can be divided into three substeps: Determination of return types of functions, determination of types of global variables, and finally determination of types of expressions.

Firstly, we describe types of expressions we want to capture and their features:

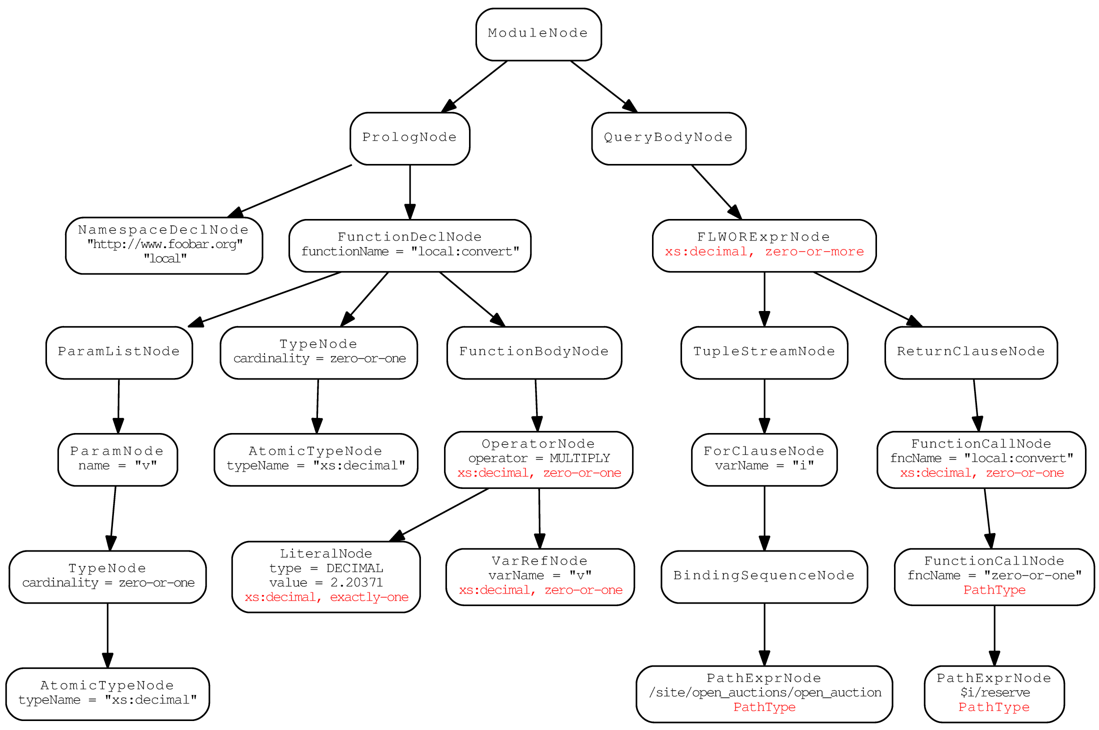

ModuleNode
PrologNode
QueryBodyNode
NamespaceDeclNode "http://www.foobar.org" "local"
FunctionDeclNode functionName = "local:convert"
FLWORExprNode xs:decimal, zero-or-more
ParamListNode
TypeNode cardinality = zero-or-one
FunctionBodyNode
TupleStreamNode
ReturnClauseNode
ParamNode name = "v"
AtomicTypeNode typeName = "xs:decimal"
OperatorNode operator = MULTIPLY xs:decimal, zero-or-one
ForClauseNode varName = "i"
FunctionCallNode fncName = "local:convert" xs:decimal, zero-or-one
TypeNode cardinality = zero-or-one
LiteralNode type = DECIMAL value = 2.20371 xs:decimal, exactly-one
VarRefNode varName = "v" xs:decimal, zero-or-one
BindingSequenceNode
FunctionCallNode fncName = "zero-or-one" PathType
AtomicTypeNode typeName = "xs:decimal"
PathExprNode /site/open_auctions/open_auction PathType
PathExprNode $i/reserve PathType

Figure 6.14: Sample syntax tree (after the static analysis of types)

- XML Schema simple data types [68].
- Types *ElementType*, *AttributeType*, *NodeType*, *TextNodeType*, *CommentType*, *ProcessingInstructionType*, and *DocumentType* representing an element, attribute, node, text node, comment, processing instruction, and document node respectively.
- Type representing a node or a set of nodes selected by a certain path expression. The path expression is included in this type. Let this type be denoted as *PathType*.
- *UnknownType* representing a type without known details, which does not suit one of the three previous types. An example is XSD type `anyType`.

PathType contains additional information. The represented path is contained by a list of its steps, in particular instances of `StepExprNode`. If a step is a reference to a variable whose type is PathType, we also want to include this information. Therefore, PathType contains an association between the steps and other PathTypes and this association is defined for the PathType variable steps.

To distinguish between a common PathType selecting a set of nodes and a PathType bound to a `for` variable in a FLWOR expression, PathType structure contains a flag `isForBound`.

Additionally, PathType contains a list of special functions that were called with an argument of PathType type. The motivation is that in some cases of function calls, we want to know that the function call is performed with an instance of PathType because then, we can determine a type of the function call more precisely. Those special functions are built-in functions `data`, `min`, `max`, `avg`, `sum`, `distinct-values`, `zero-or-one`, `exactly-one`. And other may be added, when needed.

In summary, we represent PathType as a structure with the following member variables:

- `steps` – A list of `PathExprNode` instances,
- `substeps` – An association between variable-referencing steps and instances of PathType type,
- `isForBound` – Boolean flag determining if the type was bound to a `for` variable in a `for` clause,
- `specialFunctionCalls` – A list of special functions called with this instance as an argument.

To capture sequences, we assign the first two categories of types (all types except for PathType and UnknownType) with its cardinality [162]. Each of those types can be perceived as a sequence. A type representing one value or one item can be perceived as a sequence of exactly one item. The cardinality expresses one of the following five sequence types:

- An empty sequence,
- A sequence of exactly one item,
- A sequence containing zero or one item (modifier `?`),
- A sequence containing zero or more items (modifier `*`),
- A sequence containing one or more items (modifier `+`).

PathType is not assigned with the cardinality since we do not evaluate the queries, and, therefore, we cannot determine if a certain XQuery path targets zero, one or more nodes. Alike, UnknownType is neither assigned with the cardinality. Expressions of UnknownType are not utilized it the inference, therefore, their cardinality is not needed.

Determination of Function Return Types

Determination of return types of functions is necessary, because function calls can appear in various expressions. A return type of a function can be determined at the moment the analysis of expressions encounters a call of the function; however, it involves multiple transitions of the syntax tree in a search for a definition of a particular function.

Instead, the syntax tree can be searched just once, before the analysis of expressions, and return types of all functions found are stored.

Determination of Global Variable Types

A similar approach as in case of functions can be applied to determine types of global variables. Alike the functions, the global variables are defined in the prologue subsection. A type of a variable can be explicitly specified in its definition, for instance `declare variable $x as xs:byte := 12;`. If it is not, it may often be deducible from the binding expression.

Determination of Expression Types

To determine types of expressions, we use function `analysisOfExpressionTypes`. This function is called upon a binding expression of a global variable which determines the type of the binding expression, and hence, the type of the variable. The starting node (its first argument) is the node representing the query body and the variable context is empty as there cannot be any local variable valid in the body node.

We can also determine expression types in functions. To do this for a certain function, `analysisOfExpressionTypes` function has to be called with a function declaration node as the starting node. In this case, the function declaration node contains a subnode specifying function's formal arguments. These arguments are set as the variable context for the function body represented by another subnode.

Figure 6.14 shows the syntax tree after the static analysis of expression types. The types are shown in red color. Note that the node representing `zero-or-one` function call is of a PathType type, as well as its argument. It is so, because the function returns its argument unchanged, and, thus, we included the function in the special functions list in the PathType definition.

6.7.3 Step 3: Inference of Simple Data Types

In this step, the algorithm traverses the syntax tree to infer types of elements and attributes from expressions using the type information from the previous step. These two steps could be merged together but for better comprehension we present it separately.

How are the types inferred from the expression types? We do not exploit all expressions. Only expressions of a particular type are exploited. Specifically, an expression has to contain a subexpression E of PathType type (expression representing a certain element or attribute or a set of elements or attributes). In the following text, the set represented by expression E is denoted S. Another requirement is that the expression has to be either a function call or an arithmetic operation. Also other XQuery constructs can be utilized to infer simple data types, but, since the principle is similar, we focus on the two mentioned ones as the proof of the concept.

Likewise the previous step, the syntax tree is recursively searched for expressions satisfying the conditions for the type inference. A small difference is that the recursion stops at syntax tree node types `FunctionsBodyNode` and `PathExprNode`, because the processing of these nodes requires a different approach, which we do not deal with in this paper.

The output of this step is a set of statements of the form $P \to T$, where P is an instance of PathType and T is an XML Schema simple data type.

Function Calls

This case is quite straightforward. The algorithm encounters a function call and one of the arguments is a set of elements of attributes (subexpression E representing S) represented by PathType P. To determine the type of S, it is only necessary to determine the type of the corresponding formal argument from the definition of the function. The function is either a built-in one so its definition is known, or it is defined in the prologue subsection.

If the type T of the formal argument is a simple data type or its sequence, then T is also the inferred type of S. The inferred production is $P \to T$. Otherwise, no production is inferred.

Arithmetic Operations

If the operator in an arithmetic operation is one of `+`, `-`, `div`, `mod`, `*`, `/` (the class of the expression node is `Operator` and it represents one of `PLUS`, `MINUS`, `IDIV`, `MOD`, `MUL`, `DIV`) constructs, one operand is of PathType P and the type T of the other operand is one of numeric simple data types, then the inferred production is $P \to T$.

If the operator is one of <, >, <=, >=, =, != (the class of the expression node is `Operator` and it represents one of `GEN_LESS_THAN`, `GEN_GREATER_THAN`, `GEN_LESS_THAN_EQUALS`, `GEN_GREATER_THAN_EQUALS`, `GEN_EQUALS`, `GEN_NOT_EQUALS` expressions), one operand is of PathType P and the type T of the other operand is one of simple data types, then the inferred production is $P \to T$.

6.7.4 Step 4: Key Discovery

In the last but not least step, the algorithm discovers keys of elements. In particular our approach incorporates and extends the approach from paper [50]. To discover keys and foreign keys, the method utilizes element/element joins. Assume a query Q that joins a sequence of elements S_1 targeted by a path P_1 with a sequence of elements S_2 targeted by a path P_2 on a condition $L_1 = L_2$. The method is based on an assumption that each join is done via key/foreign key pair. It means it is supposed that L_1 is a key of the elements in S_1 and L_2 is its respective foreign key or vice versa.

We consider two possible cases:

(O1) L_1 is a key of elements in S_1, L_2 is a respective foreign key and it itself is not a key of elements in S_2.

(O2) L_2 is a key of elements in S_2, L_1 is a respective foreign key and it cannot be decided whether L_1 is a key of elements in S_1 or not.

For a particular join, the decision for one of the cases (O1) and (O2) is made by the form of the join. The query is searched for so-called *join patters* – a *for join pattern* and a *let join pattern* – provided in Listing 6.1 and 6.2.

Each occurrence of a join pattern is classified by application of the following rules R_1 – R_6 in this specific order. The first satisfied rule is applied. The occurrence is also assigned with a weight determining how sure the method is about the inferred production.

The pattern occurrence is considered of case (O1)

- if it is the for join pattern (R_1, weight = 1),
- if aggregation function `avg`, `min`, `max` or `sum` is applied on a target return path (R_2, weight = 1), or
- if aggregation function `count` is applied on a target return path (R_3, weight = 0.75),

where the target return paths are paths in C_R starting with $\$e_2$ (see Listing 6.1). Otherwise, the pattern occurrence is considered of case (O2) and the assigned weight depends on the number of target return paths:

Listing 6.1: For join pattern.

```
for $e_1$ in $P_1$
return
  for $e_2$ in $P_2[L_2 = \$e_1/L_1]$
  return $C_R$
```

Listing 6.2: Let join pattern.

```
for $e_1$ in $P_1$
return
  let $e_2$ := $P_2[L_2 = \$e_1/L_1]$
  return $C_R$
```

Listing 6.3: For-for join pattern.

```
for $e_1$ in $P_1$
for $e_2$ in $P_2$
where $e_2/L_2 = \$e_1/L_1$
return $C_R$
```

- If the number is greater than one, the weight is one (R_4, weight = 1),
- else (the number equals zero or one) the weight is one half (R_5, weight = 0.5).

Last but not least, for the for-for join pattern 3 (see Listing 6.3), we introduce a new rule, considering the for-for join pattern 3 of case (O1) described above (R_6, weight = 0.5). The lower weight is chosen, because there is a lower probability that join of the join pattern 3 type is done via a key/foreign key pair.

To find the occurrences of the join patterns, the algorithm recursively, in pre-order, searches the syntax tree and every node representing a FLWOR expression is processed. The processing iterates through subnodes of a current node. For each found join pattern occurrence, the algorithm decides whether it is (O1) or (O2) case. Then rules R_2 and R_3 are applied using instances of PathType, while rules R_4 and R_5 use instances of `PathExprNode`. Rules R_4 and R_5 count target return paths. If they use instances of PathType, they can count one path more times, and thus, give a wrong result.

6.7.5 Conclusion

As we can see, *jInfer* enables optimization of the inference process even with a completely new type of input (in this case XQuery queries) and with regard to classical inference approaches completely new output (in this case simple data types and simple integrity constraints, i.e., keys and foreign keys). Thanks to the modularity, the author of such an extension can completely reuse the implementation of the inference method and focus on the extensions related to analysis of XQuery queries. The only modules that have to be modified (extended) are those responsible for generating OG and its expression in the target XML schema language.

6.8 Schematron Schema Inference

last but not least, we will show how the inference process can be extended towards inferring of a completely new type of output. This part corresponds to phase VII. of the inference process, although it may influence also the previous phases if the output schema language enables one to express additional information that need to be inferred first. In particular we will show an approach for inference of a Schematron [51] schema. Note that from another point of view this may mean also extension of XML Schema output constructs with `assert` and `report` elements that are directly inspired by Schematron. However, in this case we will output purely the Schematron schema and we propose a novel three-step algorithm. Its full description including technical details can be found in [163].

As we will show, Schematron may not be ideal for expressing general ordered unranked trees, since the schema may be bigger than the schemas of classical grammar-based schema languages (e.g., Relax NG, DTD). We divide the transformation process of a production into three steps:

1. *generate the correct context* for productions,
2. control the correct *sum of children*,
3. match the *order of children* to the RE of the production.

6.8.1 Step 1. Context Generation

The correct context for productions is absolutely necessary for the algorithm to work correctly. The context is used to match an element in an XML document to a production $\vec{h}$ from an OG G. The context is used for constraints generated by steps 2 and 3.

Trivial Solution Let us have a grammar $G = (N, T, S, P)$ and a production $\vec{h}$ of the form $A \rightarrow aR$ where $h \in P, A \in N, a \in T$ and R is a RE over N. The simplest solution is to generate the context using only the element itself. This method may work if and only if there exists an inverse function to function $trans : N \rightarrow T$ that assigns each non-terminal $A \in N$ with terminal $a \in T$ on the basis of productions from P. In that case, we can create a simple XPath expression for each production $\vec{h}$ using only the name of terminal of the production $\vec{h}$.

If we want to use relative context, there must not exist two different productions with the same terminal, i.e., in the whole XML document, we must be able to identify a production based only on the name of the element. This is, however, satisfied only by LTGs, i.e., DTDs.

For example, for non-terminal *Person* the context is `//person`.

K-ancestors This method is used for schema inference in [12]. The key idea is to identify the context on the basis of element name and the name of K closest ancestors. (Note that the trivial solution is a special case where $K = 1$.) Similarly to trivial context generation, K-ancestor solution offers fast and comfortable way to identify the context. On the other hand, K-ancestor solution may not identify some contexts correctly. However, the real world data [12] show that more than 98% of context matching could be expressed with this solution and the K equal to 2 or 3.

For example, for $K = 2$ and non-terminal *Person* the context is

```
//database/person.
```

Absolute Path without Recursion A more reliable and less restricted way to identify the correct context can be specified using absolute paths. This approach is sufficient for general cases of grammars that do not contain *recursion* or only so-called *simple recursion*.

Definition 24. *We denote the* derivation sequence D_A *for non-terminal* A *to be a sequence of non-terminals produced by productions that transform the starting non-terminal to the non-terminal* A*.*

Definition 25. *We denote* derivation sequence of terminals DT_A *for a non-terminal* A *to be a sequence of terminals defined by the formula* $\forall X \in D_A : trans(X)$*.*

In other words, the derivation sequence of terminals DT_A is a derivation sequence D_A "translated" to terminals. Then, when we want to find a context for a production $\vec{h}$ of the form $A \to aR$, we can exploit the sequence DT_A of terminals. This sequence contains terminals that match the absolute path from the root element (that matches the terminal of the starting symbol) to terminal (element) a of the production $\vec{h}$. We join the derivation sequence of terminals with character "/". For example, consider the derivation sequences: $D_{Data} = \langle Database, Person, Data\rangle$ and $DT_{Data} = \langle database, person, data\rangle$. Then, the context for non-terminal *Data* is `/database/person/data`.

Note that the derivation sequence is always deterministic. If there is a non-deterministic step (e.g., operator "?" or "|"), we generate all possible deterministic sequences and merge all their results.

Simple Recursion in Productions There can be situations when the length of a derivation sequence is not limited. This happens when there is a recursion in productions.

Definition 26. *We say that the derivation sequence* D_A *contains* recursion *if there is at least one non-terminal* $X \in D_A$ *that occurs more than once in* D_A*. We say that the recursion is a* simple recursion *if there are no other non-terminals between any two occurrences of* $X \in D_A$*.*

In other words, simple recursion is a sequence where the repetition of a single symbol is not interrupted by any other symbol (which is a common case in real-world XML data [14]). Multiple derivation sequences may exist for the same non-terminal. Without loss of generality, we suppose there exists only one such sequence. If more sequences exist, we can always merge their generated path expressions.

Definition 27. *We denote the* derivation regular expression DR_A *for a non-terminal* $A \in N$ *to be a word over* N *that represents all derivation sequences of* D_A*.*

DR_A is able to express several derivation sequences with a single finite word.

Definition 28. *We denote the* derivation regular expression for terminals DTR_A *for a non-terminal $A \in N$ to be a word over terminals T of grammar G converted from DR_A by the formula:*

$$\forall X \in DR_A \wedge X \in N : trans(X)$$
$$\forall X \in DR_A \wedge X \notin N : X$$

We can now re-define simple recursion using the derivation regular expression.

Definition 29. *We say that derivation regular expression DR_A contains only* simple recursion *if and only if all regular operators $+$ and $*$ in DR_A are applied to a single symbol (and not to a group).*

Since x^+ can be expressed as xx^*, without loss of generality we can assume that all simple recursions consist of the form x^*.

Definition 30. *Let us have a derivation regular expression DR_A and an XML fragment F. We denote* foreign elements $foreign(DR_A, F)$ *to be such elements that have to be removed from F in order to be matched by the DR_A.*

Now we introduce the algorithm for XPath context generation for a grammar with simple production recursion. Since XPath 1.0 does not support REs, we have to use the `descendant-or-self` axis with a constraint on the ancestors. We create a constraint that will ensure that we will find only such descendants that have no other element than the element from the simple recursion. The input is the DRT_A for the simple recursion.

The constraint is implemented using the `count()` function, and `ancestor` and `descendant` axes. First, we store into two variables numbers of ancestors, namely any ancestor (`allCount`) and ancestors with name x (`xCount`). We can use the `let` construct for creating these variables. We then generate the XPath expression for context ad folows:

```
//x[(count(ancestor::x) - $xCount) = (count(ancestor::*) -
                   $allCount)]
```

Recursion with Deterministic Content Finally, we will introduce a general approach that allows us to match recursions with deterministic content. First, we denote the *derivation loop* to be a part of a derivation regular expression that is being repeated. We will extend the algorithm introduced for simple recursion adding several more constructs and constraints, namely:

- The *leading terminal constraint* ensures that we match the correct leading element. This constraint may not be used every time, but it is important to notice a situation when there could be multiple elements of the leading symbol.
- The *restriction constraint* checks that there are no foreign elements.
- The *completeness constraint* checks that all terminals from derivation terminal regular expression are matched to elements within the recursion, thus no elements are missing and all elements are in the correct order.

Firstly, we have to identify the leading terminal `t`. We count its ancestors and store this value to variable `leadCnt`. The occurrence of `t` in the recursion loop is stored in variable `leadMod`. The variables are used in the generated production as follows:

```
(count(ancestor::t) - leadCnt)
        mod leadMod = 0
```

Secondly, we have to check the restriction constraint. The algorithm for simple recursion checks only a single element. We enhance it to support checking of multiple elements. For each terminal `t` in DRT_A, we compute the occurrence of `t` in ancestor nodes. For example the second phase of the algorithm generates for $DRT_A = (abcb)^*$ the following predicate:

```
//a[(count(ancestor::a) + count(ancestor::b) + count(ancestor::c)) =
                        count(ancestor::*)]
```

To check the internal structure of a recursion loop, we use nested predicates with `child` axes. We check the child, grandchild, great-grandchild, etc. of the leading terminal. We denote this condition as the *structural check*. The generated constraint will check the number of ancestor leading terminals against the number of elements found by the structural check:

```
leading_symbol[child[grand-child[ ...[leading_symbol]]]]
```

The final parent-child check matches the leading symbol of the following recursion loop. We count the number of occurrences of the leading terminal in the loop (stored as `LeadingSymbolCnt`). Since the loop is deterministic, we can check the internal structure using a single nested XPath condition. For example, if we transform loop `abcad` to XPath expression `a[b[c[a[d[a]]]]]`, the generated condition will be the following:

```
count(ancestor::a)=LeadingSymbolCnt *
  count(ancestor::a[b[c[a[d[a]]]]])
```

The DRT_A for the loop is "*abcad*", the leading symbol is thus *a* and the constant `LeadingSymbolCnt` expresses the number of symbols of the leading terminal within the recursion loop. Here it is equal to 2.

6.8.2 Step 2. Boundary Productions

From now, the found context will be denoted as `CONTEXT`. Within it we can now focus on validation using minimum and maximum occurrence checks – so-called *boundary productions* – of elements from the RE part *R* of a production $\vec{h}$ of the form $A \to aR$. We can detect elements that are not present in *R* and elements with an invalid occurrence. We process *R* and for each non-terminal X from set *S* of non-terminals in R, we count the `minOccurs` and `maxOccurs`. Both functions are defined as follows:

$$\texttt{minOccurs:}\ N \to \{0, 1, 2, ...\}$$
$$\texttt{maxOccurs:}\ N \to \{0, 1, 2, ...\} \cup \{unbounded\}$$

where N is the set of all non-terminals.

First we check that there are no illegal children with a single production:

$$\sum_{s \in S} count(trans(s)) = count(child :: *)$$

Next, we generate a production for each non-terminal from S to check the bounds:

$$\bigvee_{s \in S} count(trans(s)) \geq minOccurs(s) \wedge count(trans(s)) \leq maxOccurs(s)$$

Note that we may skip the maximum bound check, if `maxOccurs` = `unbounded`. The same fact can be applied to `minOccurs` $= 0$.

6.8.3 Step 3. Order Check

The basic idea for checking order is taken from [52]. In particular, we have a RE R and element e. We want to create a set of productions to test the order of child elements of e with regard to R. We process R sequentially from left to right. For each part of R we create constraints for allowed following siblings.

RE operators, except the grouping operator, can be expressed using conditions on following siblings. Based on the cardinality of the following (or preceding) siblings, we may use more than just one condition. For example, the derived assert rules for RE $x + yz?$ are as follows:

```
<rule context="CONTEXT/x">
  <assert test=
         "(not(preceding-sibling::*)
             or
           preceding-sibling::*[1][self::x])
          and
          (following-sibling::*[1][self::x]
             or
           following-sibling::*[1][self::y])">
    ...
  </assert>
</rule>
<rule context="CONTEXT/y">
  <assert test="following-sibling::*[1][self::z]
                or
                not(following-sibling::*)">
    ...
  </assert>
</rule>
<rule context="CONTEXT/z">
  <assert test="not(following-sibling::*)">
    ...
  </assert>
</rule>
```

6.8.4 Conclusion

The last but not least demonstration of features and advantages of *jInfer* is related to another kind of output. In this case the author of such kind of extension can just implement new module for expressing OG in the target XML schema language. All the other parts, parsing modules, visualization libraries, etc. can be reused without any change.

6.9 Open Problems

As we have shown, *jInfer* enables one to implement new improvements and extensions of the inference process. In this section we discuss a number of open problems to be solved and where *jInfer* could be successfully applied.

User Interaction In most of the existing papers the approaches focus on purely automatic inference of an XML schema. The problem is that the resulting schema may be highly unnatural. Although, e.g., the MDL principle evaluates the quality of the schema using a realistic assumption that it should tightly represent the data and, at the same time, be concise and compact, user preferences can be quite different. Hence, a natural improvement may be exploitation of user interaction. Some of the existing papers (e.g., [47]) mention the aspect of user interaction, typically in phase IV. of refinement of the result, but there seems to be no detailed study and, in particular, respective implementation. And, naturally, this problem is closely related to a suitable user interface which does not require complex operations and decisions.

In paper [152] we have proposed our preliminary attempts towards exploitation of user interaction; however, we can certainly go even further. For instance, the user may influence the merging phase by proposing preferred merging operations/target constructs, clustering similar elements etc. Such approach will not only enable one to find more concise result, but to find it more efficiently as well. Even though the solution seems to be simple, its implementation is not, since we cannot expect the user to make too many decisions or to evaluate too many choices. Hence, such a solution requires a kind of "recorder" which is able to learn from previous decisions and use them in similar cases.

Other Input Information In all the existing works the XML schema is inferred on the basis of a set of positive examples, i.e., XML documents that should be valid against the inferred schema. As we have mentioned, the Gold's theorem highly restricts the existing solutions and, hence, the authors focus on heuristic approaches or limit the methods to particular identifiable classes of languages. But another natural solution to the problem is to exploit additional information, such as an XML schema or XML queries.

In Section 6.6 we have proposed a preliminary solution exploiting an obsolete schema, but the exploitation strategy can go even further. An inspiration can be found in the area of *schema evolution* [164, 165] or correction of XML data [166, 167, 168] at a much sophisticated level.

In case of exploitation of XML queries the motivation is similar though more obvious. In Section 6.7 we have shown an approach which enables one to extend the inference process with inference of simple data types and XML keys/foreign keys which results from our previous work [50]. But, in general, the queries restrict parts of the data structure (those that should appear at output) and this partial information can be exploited for schema inference.

In addition, there seems to be no approach that would exploit negative examples (i.e., XML documents that should not conform to the schema). In this case we can find a real-world motivation again in the area of data evolution and versioning.

XML Schema Simple Data Types One of the biggest advantages of the XML Schema language in comparison to DTD is its wide support of simple data types [68]. It involves 44 built-in simple data data types such as, e.g., `string`, `integer`, `date` etc., as well as user-defined data types derived from existing simple types using `simpleType` construct. In enables one to derive new data types using *restriction* of values of an existing type (e.g., a string value having length greater than two), *list* of values of an existing type (e.g., list of integer values) or *union* of values of existing data types (e.g., union of positive and negative integers). Hence, a natural improvement of the existing approaches is a precise inference of simple data types. Unfortunately, most of the existing approaches omit the simple data types and consider all the values as strings. Two exceptions are proposed in [169, 170], but both the algorithms focus only on selected simple data types.

XML Schema Advanced Constructs The second big advantage of the XML Schema language are various complex constructs. The language exploits object-oriented features, such as user-defined data types, inheritance, polymorphism, i.e., substitutability of both data types and elements etc. Although most of these constructs do not extend the expressive power of XML Schema in comparison to DTD [171], they enable one to specify more user-friendly and, hence, realistic schemas. Naturally, their usage is closely related to the previously described problem of user-interaction, since the user can specify which of the constructs are preferred. In [172, 82, 152] we have proposed several preliminary approaches towards inference of unordered sequences, shared fragments, or type inference. But, the language itself provides much stronger tools.

Integrity Constraints As we have mentioned, both DTD and XML Schema enable one to specify not only the structure of the data, but also various semantic constraints. Both involve `ID` and `IDREF(S)` data types that specify unique identifiers and references to them. The XML Schema language extends this feature using `unique`, `key` and `keyref` constructs that have the same purpose but enable one to specify the unique/key values more precisely, i.e., for selected subsets of elements and/or attributes and valid within a specified area. In addition, the `assert` and `report` constructs enable one to express specific constraints on values using the XPath language. The current works focus mainly on the `ID`, `IDREF(S)` attributes [173, 174] and exploit various data mining approaches to find the optimal sets of keys and foreign keys. Unfortunately, all the existing works infer the keys separately, i.e., regardless a possibly existing XML schema or on the basis of an inference approach. Similarly, none of them focusses on any of the advanced constraints of XML Schema or Schematron. In addition, there are also more complex XML integrity constraints [175] that could be inferred, though they cannot be expressed in the existing schema specification languages so far, functional dependencies [176, 177] or even languages for expressing any integrity constraint in general, such as, e.g., *Object Constraint Language (OCL)* [178]. A detailed study of XML integrity constraints can be found in [179], whereas their inference would extend the optimization of approaches that analyze and exploit information on XML data from XML schemas.

Other Schema Definition Languages The DTD and XML Schema are naturally not the only languages for definition of structure of XML data, though they are undoubtedly the most popular ones. The obvious reason is that these two have been proposed by the W3C, whereas DTD is even a part of specification of XML. Nevertheless, there are also other relatively popular schema specification languages. The two most popular ones, RELAX NG and Schematron, are briefly introduced in Section 6.2. As defined in Section 6.4, the former language has higher expressive power than XML Schema and DTD, since it enables one, e.g., to combine elements and attributes in the REs. The latter one exploits completely different approach (since it is a pattern-based, not grammar-based language) and, hence, it will require completely different inference approach. The solution we have introduced in Section 6.8 is probably the first work in this area indication further possible improvements.

6.10 Conclusion

In general, the XML schema of XML documents is currently exploited mainly for two purposes – data-exchange and optimization. In the former case we usually need the inferred schema as a candidate schema further improved by a user using an appropriate editor or in cases when no schema is available. In the latter case the approaches exploit the knowledge of the schema, i.e., expected structure of the data, for optimization purposes such as, e.g., finding the optimal storage [79] or compression [180] strategy. However, in general, almost any approach that deals with XML data can benefit from the knowledge of their structure, i.e., XML schema. The only question is to what extent.

The aim of *jInfer* project is to provide a "playground", where researchers can implement and test their optimizations for particular phases of the inference process while (a) they can exploit the existing unchanged parts and (b) they are provided with all the related tools such as data parsing, visualization of the automata, transformation of the automata to XML schema languages etc.

In our future work we will focus mainly on proving the concept of *jInfer*, i.e., proposal of own optimization approaches to XML schema inference and their verification using *jInfer*. Namely we are currently dealing with inference of integrity constraints and optimization of the inference process using other input information, in our case XML queries expressed in XPath [28] or XQuery [21]. At the same time we will focus on further improvements of the implementation of *jInfer* itself, such as the GUI, support for other data inputs and outputs, etc. These aspects are less interesting from the research point of view; however, they will provide the researchers with more robust and use-friendly tool to test their proposals.

7. Conclusion

The aim of this thesis was to illustrate various steps of data analysis and processing. On this account, we have discussed the main problems and the current state of the art which comprises many different tools for data acquisition, document integration, analysis, and visualization of the results. In the following paragraphs we discuss the main contributions corresponding to three general and extensible frameworks – *Analyzer*, *Strigil*, and *jInfer* – we have proposed in this thesis and implemented in recent years, remaining open problems, and our future work.

7.1 Document Analysis

In Section 2, we introduced a complex, open and extensible system for document analysis called *Analyzer*. We described the advantages and problems of existing solutions and on the basis of these findings we designed and implemented a universal framework for batch data analysis. It allows processing of documents from different sources that consists of the following phases:

- a Web crawling and a document acquisition step,
- a data correction step,
- an analytic processing step,
- an aggregation step and a visualization of results.

We implemented and tested modules for analysis of XML documents, schemas and XQuery programs and presented the results in Section 2. Although *Analyzer* was originally devoted to XML technologies, we implemented an application that is capable of performing analyses over documents of any type. To confirm this claim, in Section 3, we presented an analysis of real-world RDF triples. We proposed several complex metrics to describe linkage between entities and compared the data sets.

Contributions The key contributions of this part of the thesis are as follows:

- We designed and implemented a universal framework which gives a user an environment for Web crawling, configuring, managing and scheduling analyses, and browsing the generated results (Section 2).
- As the first use case we implemented and tested modules for analyses of real-world XML documents and XML schemas (Section 2.6).
- We described a novel analysis of XQuery queries focused on the construct usage (Section 2.7 and Section 4).
- We proposed new metrics for publicly available Linked Data data sets and implemented an analytic module. We computed results of these characteristics over more than 20 millions of triples (Section 3).

Future Work The current analytical plugins are able to analyze the most common structural aspects of XML documents and XML schemas. Naturally they can be further extended so that they cover most of the statistics used in the related work. We can go even further and analyze new, or advanced features of *XML Schema 1.1* such as, e.g., constructs `assert` and `report`, i.e., integrity constraints over the data.

Having such a robust tool for analysis of real-world data, a natural next step is to perform an extensive analysis of the current real-world data and especially their evolution. A detailed analysis that would cover all the metrics and observations from the existing papers on data analyses (as described in Section 2.8) can be performed, whereas the found differences would bear highly useful and interesting information.

On the other hand despite the fact that XML data still keep a leading role in data representation and the related XML technologies are robust and mature, there exist other important formats and data types that become more popular. A classical example are data types related to Semantic Web [126], such as ontologies [127] or Linked Data [89]. In this case we need to solve similar issues, i.e., crawling, correction, and analyses, whereas other aspects, namely evolution are even more important.

7.2 Data Extraction

In Section 5 we described a design and an implementation of *Strigil*, a data extraction tool which allows the user to specify a set of templates to cover different document layouts. We proposed our own template language, based on well-known XSLT transformations, which can be used to extract data from HTML or text documents.

Contributions The key contributions of this part of the thesis are as follows:

- We designed and implemented a modular and easily extensible tool for data extraction from the Web, i.e., HTML and text documents (Section 5).
- The framework supports distributed crawling and it is able to recognize changes in the source documents (Section 5.4), because it could be necessary to change the scraping script to cover modified structure of the document.
- We proposed a template language to define which data should be extracted. The data are connected to an ontology, so they can be easily integrated into existing information systems (Section 5.5).

Future Work As described in Section 5 there are naturally various open problems to be focussed on. Firstly, we can improve the speed of Web crawling by with the usage of advanced caching system, i.e., we can keep local copies of all documents for all scripts in the local cache, monitor the last modified time of cached documents, and re-download only modified documents. Secondly the scraping module should be able to download sources linked from the gathered downloads, e.g., construct `frames` in HTML or construct `import` in XML Schema.

There are also remaining problems specific to the source format. For example in HTML documents, we have to solve problems with dynamic parts of the document, i.e., the data generated by JavaScript or returned by different AJAX calls. Additionally, current Web design trends lead to higher visual experience, so an efficient simulation of different user interface actions is required (e.g., when a mouse move action shows a new window with additional data).

7.3 Schema Inference

Last but not least, in Section 6 we proposed *jInfer*, a modular framework for XML schema inference. The aim of the framework is to provide a "playground", where researchers can implement and test their optimizations for particular phases of the XML schema inference process while (a) they can exploit the existing unchanged parts and (b) they are provided with all the related tools such as data parsing, visualization of the automata, transformation of the automata to XML schema languages etc. We have also shown that *jInfer* enables one to implement new improvements and extensions of the inference process.

Contributions The key contributions of this part of the thesis are as follows:

- We designed and implemented a modular and universal framework that enables one to implement, test and compare new modules of the inference process (Section 6).
- We improved grammar reducing metrics based on the MDL principle (Section 6.6).
- We proposed a method which exploits an old obsolete schema as a basis for inferred grammar (Section 6.6).
- We improved the inference process with usage of new typo of input, XML queries, to determine data types of inferred elements and attributes and recognition of keys based on join constructs (Section 6.7).
- We designed and implemented a module which exports output grammar to Schematron schema language (Section 6.8).

Open Problems Despite more than ten years of existence of DTD and XML Schema, the usage of schemas in real-world XML documents is limited and the topic of schema inference is still actual. The existing approaches still lack more complex schema constructs and usage of advanced data types which could be improved with usage of different heuristics or additional input information, such as XML operations, user interaction, etc. Additionally, the inference of integrity constraints could be improved with usage of commonly used queries. And, last but not least, a novel and interesting improvement could be provided by extension of output schema languages (e.g., Relax NG).

7.4 Exploitation of the Tools and Results

Finally, we will briefly describe current exploitation of all the three tools and their relation to other projects both within and outside the *XML and Web Engineering Research Group*[1] where they originated.

[1]`http://www.ksi.mff.cuni.cz/xrg/`

Analyzer was mainly supported by the Czech Science Foundation, GAČR project no. P202/10/0573[2] and GAČR project no. 201/09/P364[3]; within both it was extensively used. The results of analyses of RDF triples are also exploited in paper [61] to identify important characteristics that can make the management of RDF data more efficient. The paper was supported by the Grant Agency of the Charles University, GAUK project no. 410511[4].

As described before, *Strigil* is a part of a more complex framework for linked data integration (see Section 5.3.1) which was designed with respect to requirements provided by the EU FP7 ICT project LOD2[5] Work Package 9a[6]. The aim of the whole framework was to provide suitable data for a prototype of matchmaking Web Services for linked commerce data in the domain of public sector contracts [64]. Currently, a new ETL[7] tool[8], which is a follow-up of the framework, is prepared in cooperation with the *Semantic Web Company*[9], Vienna, Austria. It extends mainly the *ODCleanStore* module by direct integration of extraction modules and it improves its data processing capabilities.

Last but not least, *jInfer* was also supported by the Czech Science Foundation, GAČR project no. P202/10/0573 and GAČR project no. 201/09/P364, which is reflected by its close relation to *Analyzer.* It was primarily utilized to compare and describe different inference methods, as described in paper [181]. Additionally, it is used in the area of evolution and management of complex XML applications, i.e. propagation of changes from a higher level of abstraction to a lower level. In paper [182] five levels of the model-driven architecture are utilized, each representing a different view of a complex application, and its evolution is studied in the paper. The lowest level, called *extensional level*, represents the particular instances that form the implemented system such as, e.g., the XML documents. Its parent level, called *operational level*, represents operations over the instances, and the level above, called *schema level*, represents schemas that describe the structure of the instances, e.g., XML schemas. The last two levels, *platform-independent level* and *platform-specific level*, model the domain at different levels of abstraction. When a user wants to design a new XML format (s)he proceeds in the top-down direction from the platform-independent level to the schema level. Nevertheless, the user can also integrate existing XML documents into his/her solution in the bottom-up direction, i.e., use *jInfer* to infer a schema and map the schema to a model at the platform-specific level [183, 59].

[2]*Handling XML Data in Heterogeneous and Dynamic Environments*

[3]*Management of XML Data in (Object-)Relational Databases and Related Issues*

[4]*Efficient processing of Linked Data*

[5]`http://lod2.eu/`

[6]*LOD2 for a Distributed Marketplace for Public Sector Contracts*

[7]Extract, Transform and Load

[8]`https://github.com/mff-uk/intlib`

[9]`http://www.semantic-web.at/`

Bibliography

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, *Extensible Markup Language (XML) 1.0 (Fifth Edition).* W3C, 2008. `http://www.w3.org/TR/REC-xml`.

[2] A. Sahuguet, "Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask (Extended Abstract)," in *WebDB'00*, (London, UK), pp. 171–183, Springer, 2001.

[3] B. Choi, "What Are Real DTDs Like?," in *WebDB'02*, (Madison, Wisconsin, USA), pp. 43–48, ACM, 2002.

[4] M. Klettke, L. Schneider, and A. Heuer, "Metrics for XML Document Collections," in *XMLDM Workshop*, (Prague, Czech Republic), pp. 162–176, 2002.

[5] G. J. Bex, F. Neven, and J. Van den Bussche, "DTDs versus XML Schema: a Practical Study," in *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, WebDB '04, (New York, NY, USA), pp. 79–84, ACM, 2004.

[6] L. Mignet, D. Barbosa, and P. Veltri, "The XML Web: A First Study," in *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, (New York, NY, USA), pp. 500–510, ACM, 2003.

[7] M. Kratky, J. Pokorny, and V. Snasel, "Indexing XML Data with UB-Trees," in *ADBIS'02*, (Bratislava, Slovakia), pp. 155–164, 2002.

[8] E. Rahm and P. A. Bernstein, "A Survey of Approaches to Automatic Schema Matching," *The VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[9] A. Wojnar, I. Mlýnková, and J. Dokulil, "Structural and semantic aspects of similarity of Document Type Definitions and XML schemas," *Information Sciences*, vol. 180, pp. 1817–1836, May 2010.

[10] S. Yi, B. Huang, and W. T. Chan, "XML Application Schema Matching Using Similarity Measure and Relaxation Labeling," *Information Sciences*, vol. 169, pp. 27–46, January 2005.

[11] E. M. Gold, "Language Identification in the Limit," *Information and Control*, vol. 10, no. 5, pp. 447–474, 1967.

[12] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren, "Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data," in *Proceeding of the 17th international conference on World Wide Web*, WWW '08, (New York, NY, USA), pp. 825–834, ACM, 2008.

[13] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren, "Inference of Concise Regular Expressions and DTDs," *ACM Transactions on Database Systems*, vol. 35, pp. 1–47, May 2010.

[14] I. Mlýnková, K. Toman, and J. Pokorný, *Statistical Analysis of Real XML Data Collections*. Prague, Czech Republic: Report 2006/5, Charles University, 2006.

[15] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang, "Storing and Querying Ordered XML Using a Relational Database System," in *SIGMOD'02*, (Madison, Wisconsin, USA), pp. 204–215, ACM, 2002.

[16] H. Subramanian and P. Shankar, "Compressing XML Documents Using Recursive Finite State Automata," in *Implementation and Application of Automata* (J. Farre, I. Litovsky, and S. Schmitz, eds.), vol. 3845 of *Lecture Notes in Computer Science*, pp. 282–293, Springer Berlin Heidelberg, 2006.

[17] M. Girardot and N. Sundaresan, "Millau: An Encoding Format for Efficient Representation and Exchange of XML over the Web," *Computer Networks*, vol. 33, pp. 747 – 765, 2000.

[18] D. Raggett, A. L. Hors, and I. Jacobs, *HTML 4.01 Specification*. W3C, December 1999. `http://www.w3.org/TR/html401/`.

[19] F. Manola and E. Miller, "RDF Primer," 2004. `http://www.w3.org/TR/rdf-primer/`.

[20] *Web Services Activity*. W3C, 2009. `http://www.w3.org/2002/ws/`.

[21] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simon, *XQuery 1.0: An XML Query Language (Second Edition)*. W3C, December 2010. `http://www.w3.org/TR/xquery/`.

[22] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy, "Google's Deep Web Crawl," *Proc. VLDB Endow.*, vol. 1, pp. 1241–1252, Aug. 2008.

[23] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, "XML Schema Part 1: Structures Second Edition." World Wide Web Consortium, Recommendation REC-xmlschema-1-20041028, October 2004.

[24] J. Kosek, M. Kratky, and V. Snasel, "Struktura Realnych XML Dokumentu a Metody Indexovani," in *ITAT'03*, (High Tatras, Slovakia), 2003. (in Czech).

[25] A. McDowell, C. Schmidt, and K. Yue, "Analysis and Metrics of XML Schema," in *SERP'04*, (Las Vegas, Nevada, USA), pp. 538–544, CSREA, 2004.

[26] M. Kay, "XSL Transformations (XSLT) Version 2.0," W3C recommendation, W3C, January 2007. `http://www.w3.org/TR/xslt20/`.

[27] R. Onder and Z. Bayram, "XSLT Version 2.0 Is Turing-Complete: A Purely Transformation Based Proof," in *Proceedings of the 11th International Conference on Implementation and Application of Automata* (O. H. Ibarra and H.-C. Yen, eds.), vol. 4094 of *LNCS*, (Taipei, Taiwan, August 21-23), pp. 275–276, Springer Berlin / Heidelberg, 2006.

[28] J. Clark and S. DeRose, “XML Path Language (XPath) Version 1.0,” tech. rep., W3C, Nov. 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[29] N. Bruno, N. Koudas, and D. Srivastava, “Holistic Twig Joins: Optimal XML Pattern Matching,” in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, (Madison, Wisconsin, June 3-6), pp. 310–321, ACM, New York, NY, USA, 2002.

[30] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie, *XML Query Use Cases*. W3C, March 2007. `http://www.w3.org/TR/xquery-use-cases/`.

[31] M. Rorke, K. Muthiah, R. Chennoju, Y. Lu, A. Behm, C. Montanez, G. Sharma, F. Englich, and J. Tong, “XML Query Test Suite,” 2006. `http://dev.w3.org/2006/xquery-test-suite/PublicPagesStagingArea/`.

[32] M. Svoboda and I. Mlynkova, “Linked Data Indexing Methods: A Survey,” in *On the Move to Meaningful Internet Systems:OTM 2011 Workshops*, pp. 474–483, Springer, 2011.

[33] C. Weiss, P. Karras, and A. Bernstein, “Hexastore: Sextuple Indexing for Semantic Web Data Management,” *Proc. VLDB Endow.*, vol. 1, pp. 1008–1019, August 2008.

[34] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, “Matrix "Bit" loaded: A Scalable Lightweight Join Query Processor for RDF Data,” in *Proc. of the 19th Int. Conf. on World Wide Web*, WWW ’10, (NY, USA), pp. 41–50, ACM, 2010.

[35] E. Prud’hommeaux and A. Seaborne, “SPARQL Query Language for RDF,” 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[36] T. Neumann and G. Weikum, “RDF-3X: A RISC-style Engine for RDF,” *Proc. VLDB Endow.*, vol. 1, pp. 647–659, August 2008.

[37] T. Tran and G. Ladwig, “Structure Index for RDF Data,” in *Workshop on Semantic Data Management (SemData@VLDB) 2010*, 2010.

[38] L. Ding, L. Zhou, T. Finin, and A. Joshi, “How the Semantic Web is Being Used:An Analysis of FOAF Documents,” in *Proceedings of the 38th Annual Hawaii International Conferenceon System Sciences (HICSS’05) - Track 4 - Volume 04*, HICSS ’05, (Washington, DC, USA), pp. 113–122, IEEE Computer Society, 2005.

[39] C. Bizer, A. Jentzsch, and R. Cyganiak, “State of the LOD Cloud,” March 2011. `http://www4.wiwiss.fu-berlin.de/lodcloud/state/`.

[40] T. Grigalis, “Towards Web-scale Structured Web Data Extraction,” in *Proceedings of the sixth ACM international conference on Web search and data mining*, WSDM ’13, (New York, NY, USA), pp. 753–758, ACM, 2013.

[41] X. Zheng, Y. Gu, and Y. Li, "Data Extraction from Web Pages Based on Structural-Semantic Entropy," in *Proceedings of the 21st international conference companion on World Wide Web*, WWW '12 Companion, (New York, NY, USA), pp. 93–102, ACM, 2012.

[42] W. Su, J. Wang, and F. H. Lochovsky, "ODE: Ontology-assisted Data Extraction," *ACM Trans. Database Syst.*, vol. 34, pp. 12:1–12:35, July 2009.

[43] T. Berners-Lee, "Linked Data," 2006. `http://www.w3.org/DesignIssues/LinkedData.htm`.

[44] G. J. Bex, F. Neven, and S. Vansummeren, "SchemaScope: a System for Inferring and Cleaning XML Schemas," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, (New York, NY, USA), pp. 1259–1262, ACM, 2008.

[45] G. J. Bex, F. Neven, and S. Vansummeren, "Inferring XML Schema Definitions from XML Data," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, (Vienna, Austria, September 23-27), pp. 998–1009, ACM, New York, NY, USA, 2007.

[46] A. Raman, J. Patrick, and P. North, "The sk-strings Method for Inferring PFSA," in *In Proceedings of the*, 1997.

[47] H. Ahonen, *Generating Grammars for Structured Documents Using Grammatical Inference Methods*. Report A-1996-4, Department of Computer Science, University of Helsinki, 1996.

[48] P. Grunwald, *A Tutorial Introduction to the Minimum Description Principle*. Centrum voor Wiskunde en Informatica, 2005. `http://homepages.cwi.nl/~pdg/ftp/mdlintro.pdf`.

[49] Y.-O. Han and D. Wood, "Obtaining Shorter Regular Expressions from Finite-State Automata," *Theor. Comput. Sci.*, vol. 370, no. 1-3, pp. 110–120, 2007.

[50] M. Nečaský and I. Mlýnková, "Discovering XML Keys and Foreign Keys in Queries," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, (New York, NY, USA), pp. 632–638, ACM, 2009.

[51] R. Jelliffe, *The Schematron – An XML Structure Validation Language using Patterns in Trees*. 2001. `http://xml.ascc.net/resource/schematron/`.

[52] R. Jeliffe, "Converting Models to Schematron," 2006. `http://www.oreillynet.com/xml/blog/2006/11/converting_content_models_to_s.html`.

[53] J. Stárka, M. Svoboda, J. Sochna, J. Schejbal, I. Mlýnková, and D. Bednárek, "Analyzer: A Complex System for Data Analysis," *The Computer Journal*, vol. 55, pp. 590–615, May 2012. IF: 0.785, 5-Year IF: 0.943.

[54] J. Stárka, M. Svoboda, and I. Mlýnková, "Analyses of RDF Triples in Sample Datasets," in *COLD '12: Proceedings of the 3rd International Workshop on Consuming Linked Data of ISWC '12: 11th International Semantic Web Conference* (J. Sequeda, A. Harth, and O. Hartig, eds.), vol. 905 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2012.

[55] M. Kozák, J. Stárka, and I. Mlýnková, "Schematron Schema Inference," in *Proceedings of the 16th International Database Engineering & Applications Sysmposium*, IDEAS '12, (New York, NY, USA), pp. 42–50, ACM, 2012.

[56] M. Klempa, J. Stárka, and I. Mlýnková, "Optimization and Refinement of XML Schema Inference Approaches," in *Proceedings of the 3rd International Conference on Ambient Systems, Networks and Technologies (ANT)*, vol. 10, (Niagara Falls, Canada), pp. 120–127, Elsevier, 2012.

[57] M. Mikula, J. Stárka, and I. Mlýnková, "Inference of an XML Schema with the Knowledge of XML Operations," in *Proceedings of the 8th International Conference on Signal Image Technology and Internet Based Systems (SITIS)*, pp. 433–440, IEEE Computer Society Press, 2012.

[58] J. Schejbal, J. Stárka, and I. Mlýnková, "XQConverter: A System for XML Query Analysis," in *Proceedings of the 6th International Workshop on Flexible Database and Information System Technology of DEXA '11: 22nd International Conference on Database and Expert Systems Applications*, (Toulouse, France), pp. 129–133, IEEE Computer Society Press, 2011.

[59] J. Stárka, I. Mlýnková, J. Klímek, and M. Nečaský, "Integration of Web Service Interfaces via Decision Trees," in *Innovations in Information Technology (IIT), 2011 International Conference on*, pp. 47 –52, IEEE, april 2011.

[60] M. Svoboda, J. Stárka, J. Sochna, J. Schejbal, and I. Mlýnková, "*Analyzer*: A Framework for File Analysis," in *Proceedings of the 2nd International Workshop on Benchmarking of Database Management Systems and Data-Oriented Web Technologies of the 15th International Conference on Database Systems for Advanced Applications*, vol. 6193 of *LNCS*, (Tsukuba, Japan, April 1-4), pp. 227–238, Springer Berlin / Heidelberg, 2010.

[61] M. Svoboda, J. Stárka, and I. Mlýnková, "On Distributed Querying of Linked Data," in *Proceedings of the 11th Workshop DATESO 2012*, pp. 143–150, MATFYZPRESS, 2012.

[62] J. Stárka, M. Svoboda, J. Schejbal, I. Mlýnková, and D. Bednárek, "XML Document Correction and XQuery Analysis with Analyzer," in *Proceedings of the 10th Workshop DATESO 2011*, (Ostrava – Poruba, Czech Republic), pp. 61–72, VSB – Technical University of Ostrava, 2011.

[63] M. Klempa, M. Kozák, M. Mikula, R. Smetana, J. Stárka, M. Švirec, M. Vitásek, M. Nečaský, and I. Holubová, "jInfer: a Framework for XML Schema Inference," *The Computer Journal. (Note: paper under review process).*

[64] M. Nečaský, J. Klímek, T. Knap, J. Mynarz, J. Stárka, and V. Svátek, "Linked Data Support for Filing Public Contracts," *Computers in Industry, Special issue on New trends on E-Procurement applying Semantic Technologies*, 2013. *(Note: paper under review process).*

[65] J. Stárka and I. Holubová, "Strigil: A Framework for Data Extraction," *ODBASE 2013*, 2013. *(Note: paper under review process).*

[66] M. Klempa, M. Mikula, R. Smetana, M. Švirec, and M. Vitásek, jInfer *XML Schema Inference Framework.* `http://jinfer.sourceforge.net`.

[67] M. Necasky, "XSEM: A Conceptual Model for XML," in *Proceedings of the fourth Asia-Pacific conference on Comceptual modelling - Volume 67*, APCCM '07, (Darlinghurst, Australia, Australia), pp. 37–48, Australian Computer Society, Inc., 2007.

[68] P. V. Biron and A. Malhotra, *XML Schema Part 2: Datatypes (Second Edition).* W3C, 2004. `http://www.w3.org/TR/xmlschema-2/`.

[69] J. Clark, *XSL Transformations (XSLT) Version 1.0.* W3C, November 1999. `http://www.w3.org/TR/xslt`.

[70] D. Booth and C. K. Liu, *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer.* W3C, June 2007. `http://www.w3.org/TR/wsdl20-primer/`.

[71] J. Ferraiolo, J. Fujisawa, and D. Jackson, *Scalable Vector Graphics (SVG) 1.1 Specification.* W3C, 2003. `http://www.w3.org/TR/2003/REC-SVG11-20030114/`.

[72] D. Beckett, "RDF/XML Syntax Specification (Revised)," 2004. `http://www.w3.org/TR/rdf-syntax-grammar/`.

[73] Oracle, *OpenOffice.org Project.* Oracle, 2010. `http://www.openoffice.org/`.

[74] J. Stárka, M. Svoboda, J. Sochna, and J. Schejbal, Analyzer *1.0.* Charles University in Prague, Czech Republic, 2010. `http://analyzer.kenai.com/`.

[75] J. Sochna, *Collecting XML Data and Meta-Data from the Internet (in Czech).* Master Thesis, Charles University in Prague, Czech Republic, May 2010. `http://www.ksi.mff.cuni.cz/~bednarek/dp/Sochna.pdf`.

[76] M. Svoboda, *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`.

[77] J. Stárka, *Similarity of XML Data.* Master Thesis, Charles University in Prague, Czech Republic, September 2010. `http://www.ksi.mff.cuni.cz/~mlynkova/dp/Starka.pdf`.

[78] J. Schejbal, *A System for Analysis of Collections of XML Queries.* Master Thesis, Charles University in Prague, Czech Republic, May 2010. `http://www.ksi.mff.cuni.cz/~bednarek/dp/Schejbal.pdf`.

[79] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton, “Relational Databases for Querying XML Documents: Limitations and Opportunities,” in *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB ’99, (San Francisco, CA, USA), pp. 302–314, Morgan Kaufmann Publishers Inc., 1999.

[80] K. Runapongsa and J. M. Patel, “Storing and Querying XML Data in Object-Relational DBMSs,” in *EDBT’02*, (London, UK), pp. 266–285, Springer, 2002.

[81] M. Murata, D. Lee, M. Mani, and K. Kawaguchi, “Taxonomy of XML Schema Languages Using Formal Language Theory,” *ACM Trans. Internet Technol.*, vol. 5, pp. 660–704, November 2005.

[82] O. Vošta, I. Mlýnková, and J. Pokorný, “Even an Ant Can Create an XSD,” in *Proceedings of the 13th International Conference on Database Systems for Advanced Applications*, DASFAA’08, (Berlin, Heidelberg), pp. 35–50, Springer-Verlag, 2008.

[83] M. Kratky, J. Pokorny, and V. Snasel, “Implementation of XPath Axes in the Multi-dimensional Approach to Indexing XML Data,” in *EDBT’04 Workshops*, (Heraklion, Crete, Greece), pp. 46–60, Springer, 2004.

[84] R. Bayer, “The Universal B-Tree for Multidimensional Indexing: General Concepts,” in *WWCA’97*, (Tsukuba, Japan), pp. 198–209, Springer, 1997.

[85] R. Fenk, “The BUB-Tree,” in *VLDB’02*, (Hong Kong, China), Morgan Kaufman, 2002.

[86] P. Bohannon, J. Freire, P. Roy, and J. Simeon, “From XML Schema to Relations: A Cost-based Approach to XML Storage,” in *Proceedings of the 18th International Conference on Data Engineering*, (San Jose, CA, USA, February 26 – March 1), p. 64, IEEE Computer Society, Washington, DC, USA, 2002.

[87] M. Klettke and H. Meyer, “XML and Object-Relational Database Systems – Enhancing Structural Mappings Based on Statistics,” in *Selected papers from the 3rd International Workshop WebDB 2000 on The World Wide Web and Databases*, (London, UK), pp. 151–170, Springer Berlin / Heidelberg, 2001.

[88] *NetBeans 6.8 Platform*. `http://platform.netbeans.org/`.

[89] C. Bizer, T. Heath, and T. Berners-Lee, “Linked Data – the story so far,” *Semantic Web and Information Systems*, vol. 5, no. 3, pp. 1–22, 2009.

[90] Sun Microsystems, *Java 6 Standard Edition*. `http://java.sun.com/javase/6/`.

[91] “MySQL Connector 5.1.7.” `http://dev.mysql.com/downloads/connector/j/`.

[92] “Apache Derby 10.5.1.1 Database.” `http://db.apache.org/derby/`.

[93] “H2 Database 1.1.117.” `http://www.h2database.com/`.

[94] L. Galamboš, "Egothor 1.0, Java Search Engine," 2006. `http://www.egothor.org/`.

[95] *ISO 32000-1:2008: Document management – Portable document format*. Adobe, 2008. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/~catalogue_detail.htm?csnumber=51502`.

[96] L. Xyle, "Xyleme: A Dynamic Warehouse for XML Data of the Web," in *Proceedings of the International Database Engineering & Applications Symposium*, (Grenoble, France, July 16-18), pp. 3–7, IEEE Computer Society, Washington, DC, USA, 2001.

[97] S. Ailleret, "Larbin: Multi-purpose Web crawler," 2009. `http://larbin.sourceforge.net/`.

[98] M. Cafarella and D. Cutting, "Building Nutch: Open Source Search," *Queue*, vol. 2, pp. 54–61, April 2004.

[99] D. Judd and S. Groschupf, "Bixo – A Webcrawler Toolkit," 2009. `http://bixo.101tec.com/wp-content/uploads/2009/05/bixo-intro.pdf`.

[100] P. Mayr and F. Tosques, "Google Web APIs – An Instrument for Webometric Analyses?," in *Proceedings of the 10th International Conference of the International Society for Scientometrics and Informetrics*, (Stockholm, Sweden, July 24-28), 2005.

[101] Apache Software Foundation, "Xerces Java parser," 2010. `http://xerces.apache.org/xerces2-j/`.

[102] M. Murata, *RELAX (Regular Language Description for XML)*. 2002. `http://www.xml.gr.jp/relax/`.

[103] M. Svoboda and I. Mlynkova, "Correction of Invalid XML Documents with Respect to Single Type Tree Grammars," in *Networked Digital Technologies* (S. Fong, ed.), vol. 136 of *Communications in Computer and Information Science*, pp. 179–194, Springer Berlin Heidelberg, 2011.

[104] B. Bouchou, A. Cheriat, M. H. F. Alves, and A. Savary, "Integrating Correction into Incremental Validation," in *22emes Journes Bases de Donnes Avances, BDA (informal proceedings)*, (Lille, France, October 17-20), 2006.

[105] J. C. S. Staworko, "Validity-Sensitive Querying of XML Databases," in *Proceedings of EDBT 2006 Workshops on Current Trends in Database Technology*, vol. 4254 of *LNCS*, (Munich, Germany, March 26-31), pp. 164–177, Springer Berlin / Heidelberg, 2006.

[106] U. Boobna and M. de Rougemont, "Correctors for XML Data," in *Proceedings of EDBT 2004 Workshops on Current Trends in Database Technology*, vol. 3186 of *LNCS*, (Toronto, Canada, August 29-30), pp. 69–96, Springer Berlin / Heidelberg, 2004.

[107] S. Flesca, F. Furfaro, S. Greco, and E. Zumpano, "Querying and Repairing Inconsistent XML Data," in *Proceedings of the 6th International Conference on-Web Information Systems Engineering*, vol. 3806 of *LNCS*, (New York, NY, USA, November 20-22), pp. 175–188, Springer Berlin / Heidelberg, 2005.

[108] Z. Tan, Z. Zhang, W. Wang, and B. Shi, "Computing Repairs for Inconsistent XML Document Using Chase," in *Proceedings of the International Conference on Advances in Data and Web Management*, vol. 4505 of *LNCS*, (Huang Shan, China, June 16-18), pp. 293–304, Springer Berlin / Heidelberg, 2007.

[109] M. Svoboda, "Corrector Prototype Implementation." `http://www.ksi.mff.cuni.cz/~svoboda/projects/`.

[110] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, pp. 707–710, Feb. 1966.

[111] "SAX Project." `http://www.saxproject.org/`.

[112] H. Hosoya and B. C. Pierce, "XDuce: A statically Typed XML Processing Language," *ACM Transactions on Internet Technology*, vol. 3, pp. 117–148, May 2003.

[113] ISO, *ISO/IEC 14977:1996(E), Information technology – Syntactic metalanguage – Extended BNF*, 1996.

[114] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler, *XQuery 1.0 and XPath 2.0 Formal Semantics*. W3C, December 2010. `http://www.w3.org/TR/xquery-semantics/`.

[115] X. Zhang, B. Pielech, and E. A. Rundesnteiner, "Honey, I Shrunk the XQuery!: An XML Algebra Optimization Approach," in *Proceedings of the 4th International Workshop on Web Information and Data Management*, (McLean, Virginia, USA, November 4-9), pp. 15–22, ACM, New York, NY, USA, 2002.

[116] A. Fokoue, K. Rose, J. Siméon, and L. Villard, "Compiling XSLT 2.0 into XQuery 1.0," in *Proceedings of the 14th International Conference on World Wide Web* (A. Ellis and T. Hagino, eds.), (Chiba, Japan, May 10-14), pp. 682–691, ACM, New York, NY, USA, 2005.

[117] E. Merlo, K. Kontogiannis, and J. Girard, "Structural and Behavioral Code Representation for Program Understanding," in *Proceedings of the 5th International Workshop on Computer-Aided Software Engineering*, (Montreal, Que., Canada, July 6-10), pp. 106–108, IEEE Computer Society, Washington, DC, USA, 1992.

[118] CWI, *XMark – An XML Benchmark Project*. `http://www.xml-benchmark.org/`.

[119] D. Barbosa, L. Mignet, and P. Veltri, "Studying the XML Web: Gathering Statistics from an XML Sample," in *World Wide Web*, (Hingham, MA, USA), pp. 413–438, Kluwer Academic, 2005.

[120] W3C, *The Extensible HyperText Markup Language (Second Edition)*. W3C, August 2002. `http://www.w3.org/TR/xhtml1/`.

[121] R. Bourret, "XML and Databases." http://www.rpbourret.com/xml/XMLAndDatabases.htm, September 2005.

[122] S. DeRose, R. Daniel, P. Grosso, E. Maler, J. Marsh, and N. Walsh, *XML Pointer Language (XPointer).* W3C, August 2002. http://www.w3.org/TR/xptr/.

[123] S. DeRose, E. Maler, and D. Orchard, *XML Linking Language (XLink) Version 1.0.* W3C, June 2001. http://www.w3.org/TR/xlink/.

[124] S. Gao, C. M. Sperberg-McQueen, and H. S. Thompson, *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.* W3C, 2009. http://www.w3.org/TR/xmlschema11-1/.

[125] D. Peterson, P. V. Biron, A. Malhotra, and C. M. Sperberg-McQueen, *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes.* W3C, 2009. http://www.w3.org/TR/xmlschema11-2/.

[126] W3C, *The Semantic Web Homepage.* W3C, since 1994. www.w3.org/2001/sw/.

[127] M. K. Smith, C. Welty, and D. L. McGuinness, *OWL Web Ontology Language.* February 2004. http://www.w3.org/TR/owl-guide/.

[128] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich, "Data Summaries for On-demand Queries over Linked Data," in *Proc. of the 19th Int. Conf. on World Wide Web*, WWW '10, (NY, USA), pp. 411–420, ACM, 2010.

[129] M. A. Rodriguez, "A Graph Analysis of the Linked Data Cloud," *CoRR*, 2009.

[130] L. Afanasiev and M. Marx, "An Analysis of XQuery Benchmarks," *Information Systems*, vol. 33, no. 2, pp. 155–181, 2008.

[131] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition).* Prentice Hall, 2 ed., Sept. 2006.

[132] S. Boag, "Building a Tokenizer for XPath or XQuery," 2005. http://www.w3.org/TR/xquery-xpath-parsing/.

[133] G. Klein, "JFlex - The Fast Scanner Generator for Java," 2009. http://jflex.de/.

[134] "Java Platform, Standard Edition." http://www.oracle.com/technetwork/java/javase/overview/index.html.

[135] S. E. Hudson, "CUP Parser Generator for Java," 1999. http://www.cs.princeton.edu/~appel/modern/java/CUP/.

[136] T. Furche, G. Gottlob, G. Grasso, C. Schallhart, and A. Sellers, "OXPath: A Language for Scalable Data Extraction, Automation, and Crawling on the Deep Web," *The VLDB Journal*, vol. 22, pp. 47–72, Feb. 2013.

[137] H. Elmeleegy, J. Madhavan, and A. Halevy, “Harvesting Relational Tables from Lists on the Web,” *The VLDB Journal*, vol. 20, pp. 209–226, Apr. 2011.

[138] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang, “WebTables: Exploring the Power of Tables on the Web,” *Proc. VLDB Endow.*, vol. 1, pp. 538–549, Aug. 2008.

[139] K. Shchekotykhin, D. Jannach, and G. Friedrich, “xCrawl: A High-recall Crawling Method for Web Mining,” *Knowl. Inf. Syst.*, vol. 25, pp. 303–326, Nov. 2010.

[140] T. Furche, G. Gottlob, G. Grasso, O. Gunes, X. Guo, A. Kravchenko, G. Orsi, C. Schallhart, A. Sellers, and C. Wang, “DIADEM: Domain-centric, Intelligent, Automated Data Extraction Methodology,” in *Proceedings of the 21st international conference companion on World Wide Web*, WWW ’12 Companion, (New York, NY, USA), pp. 267–270, ACM, 2012.

[141] “ScraperWiki.” `https://scraperwiki.com/`.

[142] “Visual Web Ripper.” `https://scraperwiki.com/`.

[143] *RDF/XML Syntax Specification (Revised)*. W3C, 2004. `http://www.w3.org/TR/rdf-syntax-grammar/`.

[144] T. Knap, M. Nečaský, and M. Svoboda, “A Framework for Storing and Providing Aggregated Governmental Linked Open Data,” in *Proceedings of the 2012 Joint international conference on EGOVIS*, EGOVIS’12/EDEM’12, (Berlin, Heidelberg), pp. 264–270, Springer-Verlag, 2012.

[145] C.-H. Moh, E.-P. Lim, and W.-K. Ng, “Re-engineering Structures from Web Documents,” in *Proceedings of the fifth ACM Conference on Digital libraries*, DL ’00, (New York, NY, USA), pp. 67–76, ACM, 2000.

[146] R. K. Wong and J. Sankey, *On Structural Inference for XML Data*. Report UNSW-CSE-TR-0313, School of Computer Science, The University of New South Wales, 2003.

[147] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim, “XTRACT: a System for Extracting Document Type Descriptors from XML Documents,” *SIGMOD Rec.*, vol. 29, pp. 165–176, May 2000.

[148] H. Fernau, “Learning XML Grammars,” in *Proceedings of the Second International Workshop on Machine Learning and Data Mining in Pattern Recognition*, MLDM ’01, (London, UK), pp. 73–87, Springer-Verlag, 2001.

[149] J.-K. Min, J.-Y. Ahn, and C.-W. Chung, “Efficient Extraction of Schemas for XML Documents,” *Inf. Process. Lett.*, vol. 85, pp. 7–12, January 2003.

[150] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls, “Inference of Concise DTDs from XML Data,” in *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB ’06, pp. 115–126, VLDB Endowment, 2006.

[151] J. Berstel and L. Boasson, “XML Grammars,” in *Mathematical Foundations of Computer Science*, LNCS, pp. 182–191, Springer, 2000.

[152] J. Vyhnanovská and I. Mlýnková, “Interactive Inference of XML Schemas,” in *Research Challenges in Information Science (RCIS), 2010 Fourth International Conference on*, pp. 191–202, May 2010.

[153] R. Barták, *On-Line Guide to Constraint Programming*. 1998. `http://kti.mff.cuni.cz/~bartak/constraints/`.

[154] “The NetBeans Platform.” `http://netbeans.org/features/platform/index.html`.

[155] “Module System API.” `http://bits.netbeans.org/dev/javadoc/org-openide-modules/org/openide/modules/doc-files/api.html`.

[156] M. Klempa, M. Mikula, R. Smetana, M. Švirec, and M. Vitásek, “jinfer tutorial.” `http://jinfer.sourceforge.net/doc_tutorial.html`.

[157] *Document Object Model (DOM)*. W3C, 2004. `http://www.w3.org/TR/DOM-Level-3-Core/`.

[158] “Java Universal Network/Graph Framework.” `http://jung.sourceforge.net/`.

[159] M. Klempa, “Optimization and Refinement of XML Schema Inference Approaches,” Master’s thesis, Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic, 2011. `http://www.ksi.mff.cuni.cz/~mlynkova/dp/Klempa.pdf`.

[160] P. Grünwald, *The Minimum Description Length Principle*. Mit Press, 2007.

[161] M. Mikula, “Inference of an XML Schema with the Knowledge of XML Operations,” Master’s thesis, Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic, 2012. `http://www.ksi.mff.cuni.cz/~mlynkova/dp/Mikula.pdf`.

[162] J. Schejbal, “A System for Analysis of Collections of XML Queries,” Master’s thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2010. `https://is.cuni.cz/webapps/zzp/detail/78348/?lang=en`.

[163] M. Kozák, “Schematron Schema Inference,” Master’s thesis, Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic, 2012. `http://www.ksi.mff.cuni.cz/~mlynkova/dp/Kozak.pdf`.

[164] G. Guerrini and M. Mesiti, “X-Evolution: A Comprehensive Approach for XML Schema Evolution,” in *Proceedings of the 2008 19th International Conference on Database and Expert Systems Application*, (Washington, DC, USA), pp. 251–255, IEEE Computer Society, 2008.

[165] M. Nečaský, J. Klímek, L. Kopenec, L. Kučerová, J. Malý, and K. Opočenská, “XCase – A Case Tool for Designing XML,” January 2009. `http://www.codeplex.com/xcase`.

[166] B. Bouchou, A. Cheriat, M. H. F. Alves, and A. Savary, "Integrating Correction into Incremental Validation," in *BDA* (D. Laurent, ed.), 2006.

[167] S. Staworko and J. Chomicki, "Validity-Sensitive Querying of XML Databases," in *Current Trends in Database Technology EDBT 2006* (T. Grust, H. Hapfner, A. Illarramendi, S. Jablonski, M. Mesiti, S. Meller, P.-L. Patranjan, K.-U. Sattler, M. Spiliopoulou, and J. Wijsen, eds.), vol. 4254 of *Lecture Notes in Computer Science*, pp. 164–177, Springer Berlin / Heidelberg, 2006. 10.1007/11896548_16.

[168] M. Svoboda and I. Mlýnková, "Correction of Invalid XML Documents," in *Proceedings of the 16th International Conference on Database Systems for Advanced Applications (submitted)*, DASFAA '11, Springer-Verlag, 2011.

[169] B. Chidlovskii, "Schema Extraction from XML Collections," in *Proceedings of the 2nd ACM/IEEE-CS Joint Conference on Digital libraries*, JCDL '02, (New York, NY, USA), pp. 291–292, ACM, 2002.

[170] J. Hegewald, F. Naumann, and M. Weis, "XStruct: Efficient Schema Extraction from Multiple and Large XML Documents," in *Proceedings of the 22nd International Conference on Data Engineering Workshops*, ICDEW '06, (Washington, DC, USA), pp. 81–81, IEEE Computer Society, 2006.

[171] I. Mlýnková, "Similarity of XML Schema Definitions," in *Proceeding of the eighth ACM Symposium on Document Engineering*, DocEng '08, (New York, NY, USA), pp. 187–190, ACM, 2008.

[172] I. Mlýnková and M. Nečaský, "Towards Inference of More Realistic XSDs," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, (New York, NY, USA), pp. 639–646, ACM, 2009.

[173] G. Grahne and J. Zhu, "Discovering Approximate Keys in XML Data," in *Proceedings of the eleventh international conference on Information and knowledge management*, CIKM '02, (New York, NY, USA), pp. 453–460, ACM, 2002.

[174] D. Barbosa and A. Mendelzon, "Finding ID Attributes in XML Documents," in *Database and XML Technologies a Xsym 2003*, vol. 2824 of *Lecture Notes in Computer Science*, pp. 180–194, Springer Berlin / Heidelberg, 2003.

[175] K. Opočenská and M. Kopecký, "Incox – a Language for XML Integrity Constraints Description," in *DATESO'08: Database, Text, Specifications and Objects*, (Desna – Cerna Ricka, Czech Republic), pp. 1–12, CEUR-WS.org, 2008.

[176] C. Yu and H. V. Jagadish, "XML Schema Refinement Through Redundancy Detection and Normalization," *The VLDB Journal*, vol. 17, pp. 203–223, March 2008.

[177] F. Fassetti and B. Fazzinga, "FOX: Inference of Approximate Functional Dependencies from XML Data," in *Proceedings of the 18th International Conference on Database and Expert Systems Applications*, (Washington, DC, USA), pp. 10–14, IEEE Computer Society, 2007.

[178] *Object Constraint Language Specification, version 2.0.* OMG, 2009. `http://www.omg.org/technology/documents/formal/ocl.htm`.

[179] W. Fan, "XML Constraints: Specification, Analysis, and Applications," in *Database and Expert Systems Applications, 2005. Proceedings. Sixteenth International Workshop on*, pp. 805–809, 2005.

[180] C. J. Augeri, D. A. Bulutoglu, B. E. Mullins, R. O. Baldwin, and L. C. Baird, III, "An Analysis of XML Compression Efficiency," in *Proceedings of the 2007 workshop on Experimental Computer Science*, ExpCS '07, (New York, NY, USA), ACM, 2007.

[181] I. Mlýnková and M. nečaský, "Heuristic Methods for Inference of XML Schemas: Lessons Learned and Open Issues," *Informatica*, 2012.

[182] M. Nečaský, J. Klímek, J. Malý, and I. Mlýnková, "Evolution and Change Management of XML-based Systems," *Journal of Systems and Software*, vol. 85, no. 3, pp. 683 – 707, 2012.

[183] J. Klimek and M. Necasky, "Semi-automatic Integration of Web Service Interfaces," *2012 IEEE 19th International Conference on Web Services*, vol. 0, pp. 307–314, 2010.