Xtractor: A Light Wrapper for XML Paragraph-Centric Documents

Youakim BADR
National Institute of Applied Sciences - Lyon
PRISMa - Production Engineering and Computer Science for Manufacturing Systems
PO Box 7, av. Jean Capelle 69621 Villeurbanne - FRANCE
youakim.badr@insa-lyon.fr

Abstract

The emergence of XML leads the development of applications centric XML-documents. Often the documents contain tagged paragraphs of natural language texts. The extraction of relevant data from paragraphs confronts with their irregular structure hidden in the text and requires powerful extraction patterns. Although a large spectrum of wrappers has been conceived to mainly process HTML pages, the wrappers cannot deal with semi-structured data and cannot still take into consideration the natural language processing. In this paper, we present a specification language to write expressive and easy extraction patterns by casual users in a regular expression fashion. Moreover, we introduce the Xtractor, which relies on linguistic parsing of paragraphs and applies technical and natural language dictionaries.

1. Introduction

Information Extraction addresses the problem of extracting specific information from a collection of documents formatted in HTML [1], XML [2] or natural language text [3] [4]. On the other side, wrappers are conceived to extract information from web pages and return their results as structured data tuples for mediators [5]. Many studies have been investigated to build wrappers for HTML pages [6], [7], [8] and recently for XML documents [9] as well as free texts in the natural language [10].

Besides the lacks of wrappers discussed later in the related work section, most wrappers fail to extract properly relevant data from textual information, even though the documents appear to be structured in a highly regular fashion. Delimiters, such as HTML tags, are not sufficient to trigger the extraction of data, which is hidden in a tagged text. Missing attributes, multiple attributes values, and attribute permutations make wrappers fail to deal with data of irregular structure (also called semi-structured data).

On other words, wrappers, which are originally designed to process HTML pages, cannot deal with semi-structured data. The structure is irregular and it is not easy to find some uniform syntactic clues, such as delimiters, or even linguistic knowledge to correctly extract the attributes. Moreover, designers omit the operators of their wrappers (i.e. casual users), which cannot easily maintain or customize their wrappers without an expert support.

In this paper, we present an approach and architecture to design a wrapper for a family of XML documents, called Paragraph-Centric Documents (thereafter PCD). PCDs are characterized by tagged paragraphs of free text. The motivation behind building wrappers for documentary-like applications is driven by the emergence of XML as standard exchange format and data representation. The core idea of our wrapper is threefold:

1- Process tagged paragraphs in PCDs with a light parsing in order to associate tokens with entries in linguistic and technical dictionaries.

2- Define a specification language for casual users to design easy and expressive extraction patterns in order to locate data of irregular structure.

3- Provide an abstraction level easy to understand by casual users to refer to collections of words by means of meta-words. The meta-words are defined in dictionaries and used in extraction patterns.

Coupling paragraphs parsing and writing expressiveness extraction patterns over meta-words provide a convivial tool for casual users to customize the extraction of semi-structured data in XML PCDs. In this paper, we present the specification language based on regular expressions and we illustrate the implementation of the appropriate wrapper (called Xtractor) based on text parsing.

2. Related work

Several researchers investigate approaches to build wrappers [11]. The most primitive approach to construct a wrapper consists of writing a program by hand in some programming languages (e.g. Perl or

grep). This approach is impractical because the wrapper code is hard to maintain and cannot be easily exported to different application domains. Some researches propose specialized languages to quickly construct wrappers or the concept description frame of InfoExtractor [12]. Most of these languages are expressiveness, but mastering these languages still requires computer expertise. Another class of approaches intends to automatically generate wrappers [11] [8]. To the best of our knowledge, these approaches only cover small portions of the web pages with some regularity in their structures. The work of [21] defines a family of wrappers based on linear Finite State Transducers (FSTs) [13]. In the context of wrappers, FSTs recognize well the delimiters (e.g. HTML tags) surrounding relevant data. They are also studied by the natural language community to build electronic dictionaries and perform text processing (e.g. lexical, syntax and morphological analysis). But in many cases, the wrappers based on FSTs [14] address HTML documents and rely on their tags as delimiters. In addition, they do not take into consideration their capabilities for text processing in order to extract data from natural text.
The approach, that we conceive, falls into the broad category of manual systems that take into consideration the construction of expressive extraction patterns by casual users as well as the extraction of relevant data from natural language or telegraphic text. The extraction patterns are applied to a domain of interest, which is well defined by means of electronic dictionaries. A remarkable characteristic of our approach consists of translating the extraction patterns to Finite State Transducers (FST) and then employs the FSTs to build electronic dictionaries as well as parsing the text. Recent advances in the development of sophisticated tools for building FSTs (e.g. XRCE finite state tools [15]) and in the natural language community [16] have fostered the development of complex FST systems for Natural Language Processing.

3. Study Case

Our model relies on a mixture of specification language for extraction patterns and linguistic analyzer to retrieve relevant data. An interesting case to study is the patient record. XML is a good candidate to model patient records as PCDs and gives semantic to their contents. The primary task of the wrapper is to apply extraction patterns in order to locate relevant data listed in paragraphs. The wrapper returns the results as data tuples of attributes.

For example, the Figure 1 (a) illustrates a paragraph of prescriptions in a PCD. Each prescription provides information about a sequence of attributes to locate their values. In this case, the attributes are dosage, frequency, medication, and duration.

2 pills 3 times per day of KARDEGIC during 2 weeks.
DOLIPRANE : 2 pills 3x/d during 2 weeks.
ATARAX – 2 tablets morning-noon-evening for 2 wks.
take 2 and half pills of SECTRAL 3 times, 1 pill CYSTINE B6

(a)

(2 pills, 3 times per day, KARDEGIC, 2 weeks)
(2 pills, 3x/d, DOLIPRANE, 2 weeks)
(2 tablets, morning-noon-evening, ATARAX, 2 wks)
(2 and half pills, 3 times, SECTRAL, null)
(1 pill, null , CYSTINE B6, null)

(b)

Figure 1: Medical prescriptions

A wrapper is supposed to analyze the paragraph content, locates the attribute values from each sentence and then returns a set of prescription tuples as shown in Figure 1 (b). Finding relevant data in these prescriptions requires powerful extraction patterns with a substantial attention for end-users.

4. Specification Language for Patterns

4.1. Information Extraction

Our original motivation in developing our wrapper, Xtractor, is to build a system that is more appropriate to the Information Extraction task rather than a full text understanding system. In our context, Information Extraction from XML PCD documents is a kind of documents indexing. The relevant data can be stored in a database and exploited later as a decision support.

For a given domain of interest, such as the medical domain, there can be fairly elaborate dictionaries for describing most of words in paragraphs. Often, Information Extraction reveals attributes in those dictionaries hidden in paragraphs. An important aspect of Information Extraction is the designing of extraction patterns able to describe the accurate context for relevant data. The core idea of our wrapper is to process progressively tagged paragraphs in PCDs with light parsing in order to tokenize sentences and associate tokens with lexemes in dictionaries.

4.2. Dictionaries for domains of interest

For a domain of interest, such as the medical domain, we elaborate dictionaries for describing frequent words in prescriptions. A dictionary is a flat file of entries; each entry defines a lemma followed by its canonical form and a list of meta-words, which the lemma belongs to. For example, the following entries:

Doliprane, Doliprane 200 : <medication>
Cystine, Cystine B6: <medication>, <vitamin>

denote the lemmas Doliprane and Cystine and their canonical nouns. We also notice that medication and vitamin stand for meta-words, which referred by <medication> and <vitamin> in the extraction pattern. Thus, the <medication> meta-word can be one of the lexemes Doliprane or Cystine and other medication defined in the dictionary. Often, it is as well useful to define compound words as lemma entry in dictionaries and associate it with a meta-word for a high level of abstraction (e.g. <tumor>).

We distinguish two sets of dictionaries: dictionaries of the current natural language e.g. English, and dictionaries of the current domain of interest e.g. the prescription domain. In spite of the large spectrum of frequent words, we build many dictionaries to cover the prescription domain (e.g. medications, diseases and symptoms). Afterward, dictionaries will stand for either build-in or add-in dictionaries interchangeably.

```
xmedication ="([A-Z]+)";
xdosage=".*([\\d]|(?:[\\d] (?:and half|and quarter) [\\d]) |(?:[\\d]
(?:and|or) [\\d]/[\\d]))?((?: pill|tablet|capsule)[s]?)";
xfrequency=".*(([\\d]( ?:(times|x) (per|\\/)? (day|week)[s]?) )   |
            (morning-noon-evening) | (1-1-1)).*;
xduration  =".*(?: (((?:during)|(?:for) )(?:week|wk)?[s] ?) |
             (?:per)( ?:day|d)[s]?)";
____________________________________________________________
medication = <medication>
dosage = <NB> <dosage-unit>
frequency=<NB>(times ? per |x/ )<period>|(<part-day> - ?)*
duration=(during | for | per )<time-unit>
pattern = { medication +, dosage, frequency, duration? }
```

Figure 2: Regular expressions (upper side) and their abstraction (lower side)

4.3. A Glance at Regular Expressions

Regular expressions are widely regarded as a precise and succinct notation for specifying a text search. Many people routinely use regular expressions to specify searches in text editors. Regular Expressions seem to be a good candidate for manual patterns. Relevant data can be well located by describing their context and their characteristics in regular expressions fashion. Indeed, characteristics such as multiple values, missing value and permutations can be easily simulated by using basic operators like kleene (*), optional (?), union (|) and concatenation (.). Complex regular expressions can be built up from simpler ones by means of regular expression operators and parentheses. Because regular languages are closed under concatenation and union, the basic operators can be combined with any kind of regular expression.

Practically, this solution is not elegant and convenient, and it theoretically increases the complexity of the problem. Regular expressions offer a new perspective for designing matching and extraction patterns. This perspective is particularly relevant to search texts pre-processed by linguistic tools. As we demonstrate in the next paragraph, extending regular expressions can broadly be regarded as precise and concise notations for specifying patterns in order to search pre-processed texts.

4.4. Extraction Patterns

The syntax of regular expressions can be extended by words and meta-words over text. Such convenient extension allows a concise and expressiveness syntax. As shown in Figure 2 (upper side), long regular expressions with multiple nesting levels and operators become unreadable and hard to maintain. Thus, we introduce three layers to decompose long regular expressions into modular patterns, called extraction patterns. These layers are *terms layer*, *expressions layer* and *slots layer*. Casual users use to build extraction patterns by means of layers and assign one or more extraction patterns to each tagged paragraph in the PDC.

For the sake of compactness, we define informally these layers; we point out that a future work will define formally the specification language and provide necessary preliminaries and tools. We start with the definition of each layer and we illustrate them by examples.

Definition (Terms Layer):

A *term* represents an abstraction of linguistic information over text vocabulary. A term t is either:

 A *form*: a sequence of letters delimited by double quotes e.g. “cholesterol” that matches itself.
 A *formal symbol*: is a predefined form to describe a number <NB> and any word <MOT>.
 A *meta-word*: a reference to one or more lemmas in dictionaries e.g. <medication> indicates all lemmas in medication dictionary. The declarations of meta-words are specified in the construction stage of add-in dictionaries.

Definition (Expressions layer):

The *expression layer* contains a finite set of expressions; we denote by an expression a concatenation of terms separated by separator operators (i.e. white-space and tabulation).

e.g. <NB><month><NB> and <day>

We say that an expression holds if we find a sequence of terms that matches the expression.

Definition (slots layer):

A *slots layer* is made of alternates of expressions, we denote the alternate operator by | (pipe).

For example, the following four expressions of date in one slot cover all possible date formats: <NB> <month> <NB> | <NB> '/' <NB> '/' <NB> | <day>',' <month><NB>',' <NB>|<month><NB>',' <NB>.

We say that a slot holds if one of its expressions holds. A slot identifies relevant information or an attribute value defined by different contextual delimiters.

Definition (Unary operators):

Unary operators are applied on expressions to specify occurrences or repetitions. The unary operators are: kleene (*), optional (?), and one-or-more (+). We note that the use of parentheses changes the priority.

e.g. <international-code>? "phone:" (<NB>+ "-")*

Definition (extraction pattern):

An *extraction pattern* P over a paragraph p is a finite and unordered set of slots. |P| denotes cardinality of P (i.e. number of slots). We mention that all slots in an extraction pattern occur without order. Furthermore, an extraction pattern of cardinality |P| has |P|! possible combinations of slots. As a result, an extraction pattern locates matching sequence of slots in any order.

For example, the extraction pattern of three slots (e.g. *medication*, *dosage* and *frequency*) has six permutations, it locates all possible sequences and in any order of slots.

Conditions on slots order:

To constrain the search or define sub-sequences of some slots, we introduce the condition notation by means of forbidden and allowed. In a brief, the forbidden condition on a sub-sequence of two or more slots eliminates the sub-sequence in question from the all-possible combinations of slots. In contrast to forbidden, the keyword allowed restrains the sub-sequence to appear in such order in each possible combination.

Tagging relevant data:

Since the extraction patterns are slightly regular expressions, it is convenient to mark up relevant data once the regular expression holds.

```
dictionary dictionary_name: /* one or more dictionaries */
        /* one or more expressions   */
  expression  expression_identifier =  concatenation of terms   ;
        /* one or more slots   */
  slot slot_identifier   =disjunction of  expressions;
        /* one or more patterns based on predefined slots and expressions */
 pattern pattern_identifier = unordered list of slots separated by commas ;
 allowed   =unordered list of constraints on sub-sequence of slots;
 forbbiden = unordered list of constraints on sub-sequence of slots;
```

Figure 3: The formal syntax for the specification language

To deal with this issue, we delimit relevant metadata located in text by XML tags. At the extraction pattern level, the tag name is specified by an identifier of the format TagName[term] or TagName[expression], where the TagName encloses the term or expression in the output when entirely the extraction pattern holds. Roughly speaking, the notation terms, expressions, slots, and extraction patterns is implemented as a formal syntax with similarity to procedural programming languages, where identifiers of expressions, slots and patterns are declared and defined within the scope of a dictionary of extraction patterns. The formal syntax is illustrated in Figure 3.

4.5. Translate extraction patterns to FST

Finite state techniques [17] are widely used in various areas of NLP. Regular expressions are the appropriate level of abstraction for thinking about finite state languages. Informally, a FST is a device that recognizes some sequences in the input, and associates them with some outputs. Sequences are characters or words in the text written in a natural language; outputs are some linguistic information.

The specification language is easy to understand and maintain by end-users, but it is not a computational device for computers to directly carry out a real information extraction. The missing part is to translate the semantic behind the syntax to executable computational devices. After all, an extraction pattern is a regular expression and the language defined by a pattern is a regular language (Kleene theorem [18] and an equivalent finite state automaton can be generated that recognizes this language; Thomson theorem [22].

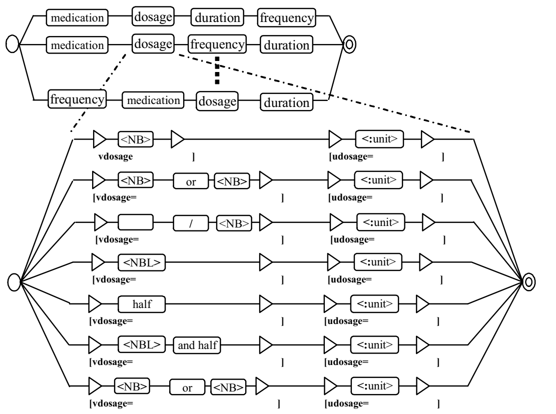

Figure 4: Equivalent FST for dosage and the FST of the extraction pattern

By using an FST, all possible combinations defined in an extraction pattern can be illustrated; at the expression level, it is not tricky to find an elementary FST as a sequence of nodes, where each node represents a term of the expression. At the slot level, the slot is presented by a FST of disjunction of paths, where each path contains a single node representing an expression of the slot. Thus, the extraction pattern becomes a very huge as illustrated in the lower part of the Figure 4 and it is even hard to be generated using a graphic tool.

5. The wrapper: Xtrator

The Xtractor wrapper applies specific domain dictionaries and linguistic analyzers to paragraphs and then employs the FSTs of extraction patterns to locate relevant data. More details about the Xtractor architecture is depicted in Figure 5. the Xtractor is organized into two major components, which are further broken down into smaller logical sub-components. The major components of Xtractor are: Linguistic Parser and Pattern Locator. The main role played by each of the sub-components is described below.

- **Text pre-processing:** At the beginning, the Xtractor reads documents. It checks that the content conforms to a valid text encoding system (ANSI 256 characters). The main goal of this sub-component relies on scanning paragraphs to recognize tokens such as simple forms.

- **Text Analysis:** After having pre-processed paragraphs, the text analysis segments each paragraph into sentences by recognizing boundaries. Then, it identifies and marks unambiguous words. This operation corresponds to a look-up the linguistic dictionaries. Finally, it detects and marks special tokens, such as elided, contracted words, and unambiguous abbreviations, etc.

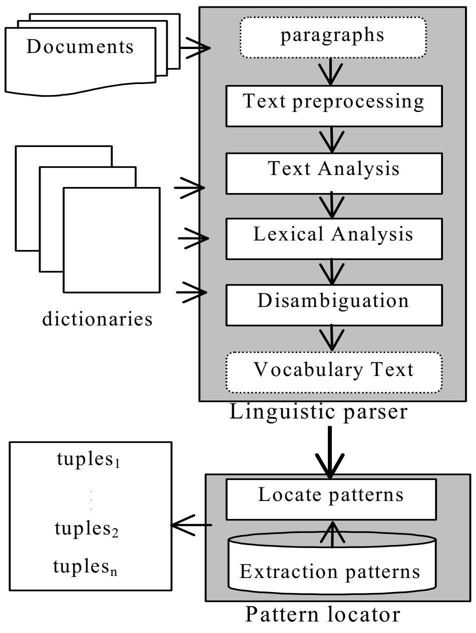

Figure 5 : The Xtractor Architecture

- **Lexical Analysis:** After the completion of the text analysis, a set of dictionaries is applied. As building dictionaries from draft is a tedious task, we adopt LADL [16] dictionaries format, we exploit dictionaries in the format of DELAF for simple forms and of DELACF for compound forms. As a result of lexical analysis, all words in paragraphs are accessible in various ways.

- **Disambiguation:** Applying resources to paragraphs may introduce ambiguity. Ambiguous words are words that correspond to more than one entry in the dictionaries. We limit our disambiguation sub-component to apply dictionaries, which contain all deviant unambiguous compound words.

The application of all sub-components results in a *text vocabulary* indexed by dictionaries. In the last step, the Pattern Locator is invoked with text vocabulary in argument.

5.2. Pattern Locator

The core of Xtractor is the Pattern Locator component. This component reads a text vocabulary of a particular paragraph delivered by the linguistic parser, and loads its appropriate extraction pattern. The Pattern Locator translates the syntax of the extraction pattern to an equivalent FST. Thus, the FST is applied to recognize matching sequences and marked up relevant features with open and close delimiters.

6. Implementation

A primary release of the Xtractor is implemented; a compiler is developed using JavaCC to recognize the formal syntax of the specification language. The compiler also implements the conversion algorithm of Thomson [22] to generate the appropriate FST for each extraction pattern. In the prototype, we inherit from Unitex [19] its capabilities to build Finite State descriptions to implement the Linguistic Parser. The Pattern Locator is built around the OraMatcher package [20] to extract the matched data after applying the FSTs of extraction patterns. We adopted ADL dictionaries format to describe our working language and get advantage of available dictionaries.

7. Conclusion and future work

In this paper, we presented an approach and architecture to design a wrapper, called Xtractor, for a family of XML Paragraph-Centric Documents. The Xtractor is based on linguistic parsing to mark up words in paragraphs against meta-words in electronic dictionaries. We defined a high-level specification language based on regular expressions to write extraction patterns. Coupling paragraphs parsing and writing expressiveness extraction patterns provide a convivial tool for casual users to customize the extraction of semi-structured data.

At the present, we are applying Xtractor to corpus of French language. To describe our specific domain of application, we built small dictionaries. The initial experiment results show satisfactory performance. We are currently building a corpus of prescriptions and stating respectively precision and call ratio in order to compare our work to the state of the art. Furthermore, we are working on the automatic induction of extraction patterns, which is mixing the linguistic approach and finite state transducers with some prior learning algorithms.

8. References

[1] Crescenzi V., Mecca G. and Merialdo P. RoadRunner: Towards Automatic Data Extraction from Large Web Sites, 27th International Conference on Very Large Databases (VLDB 2001)

[2] Muslea I. Extraction patterns for information extraction tasks: A survey. In AAAI-99 Workshop on machine learning for information extraction, 1999.

[3] Douglas A., Hobbs J., Bear J., Israel D., and Tyson M. FASTUS: A Finite-State Processor for Information Extraction from Real-World Text, Proceedings. IJCAI-93, Chambery, France, August 1993.

[4] Poibeau T. A corpus-based approach to Information Extraction, In Journal of Applied System Studies, vol. 2 n°2, 2000.

[5] Bressan, S. and Bonnet Ph. : Extraction and Integration of Data from Semi-structured Documents into Business Applications Conference on the Industrial Applications of Prolog 1997.

[6] Kushmerick N. Wrapper Induction for Information Extraction, PhD Dissertation, University of Washington, 1997.

[7] Kushmerick, N. Finite-state approaches to Web information extraction. In Proc. 3rd Summer Convention on Information Extraction, Rome 2002.

[15] Karttunen L., Chanod J-P., Grefenstette G. and Schiller A. Regular Expressions for language Engineering. Journal of Natural Language Engineering vol 2 no 4 (1997) pp 307-330, 1997 Cambridge University Press ISSN:1351-3249

[10] Piskorski J. and Neumann G. An Intelligent Text Extraction and Navigation System In proceedings of 6th International Conference on Computer-Assisted Information Retrieval (RIAO-2000), Paris, 2000

[8] Hsu C-N. Initial results on wrapping semistructured web pages with finite-state transducers and contextual rules. In Proceedings of AAAI-98 Workshop on AI and Information Integration, Technical Report WS-98-01, Menlo Park, CA, 1998. AAAI Press.

[9] Liu L., Pu C. and Han W.XWrap: An XML-enabled Wrapper Construction System for Web Information Sources, Proceedings of the Sixteenth International Conference on Data Engineering March, 2000, San Diego, CA (IEEE CS Press).

[11] Kuhlins S. and Tredwell R. Toolkits for Generating Wrappers: A survey, In Net.ObjectDays, Erfurt, Germany, September 2002.

[12] Smith D.J. and Lopez M. (1997): Information extraction for semi-structured documents, Proc. Workshop on Management of Semi-structured Data, May 1997

[13] Eilenberg S. "Automata, Languages and Machines". New York: Academic Press [online]. 1974, vol A. ISBN 0122340019.

[14] Hsu C-N. and Dung M.T. Generating finite-state transducers for semistructured data extraction from the web. Information Systems, 23(8):521-538, Special Issue on Semistructured Data, 1998.

[16] Gross M. "The Use of Finite Automata in the Lexical Representation of Natural Language". In Electronic Dictionaries and Automata in Computational Linguistics. Berlin: Springer-Verlag, 1989, vol 377, p.34-50.

[17] Karttunen L., Koskenniemi K. and Noord G. "Finite State Methods in Natural Language Processing". Natural Language Engineering. 2003, vol 9, n°1. p.1-3.

[18] Martin C. "Introduction to languages and the theory of computation". 2nd edition. New York: The McGraw-Hill Companies, 1997, 450 p. ISBN 0070408459

[20] Original Reusable Objects. "OROMatcher 1.0 User's Guide"[online].http://www.savarese.org/oro/docs/OROMatcher/ (last visited 11/10/2003)

[19] Unitex Unitex Home page [online] http://www-igm.univ-mlv.fr/~unitex/index (last visited 26/8/2004)

[21] Ashish N. and Knoblock C. (1997). Wrapper Generation for Semi-structured Internet Sources. ACM SIGMOD Workshop on Management of Semi-structured Data, 1997, Tucson , Arizona

[22] Hopcroft E.J. and Ullman D.J. "Introduction to automata theory languages, and computation". 1st edition. USA: Pearson Education POD, 1979, 418 p.