

DRUID: COUPLING USER WRITTEN DOCUMENTS AND DATABASES

Youakim Badr, Frédérique Laforest, André Flory
*Laboratoire d'Ingénierie des Systèmes d'Information,
 Institut National des Sciences Appliquées de Lyon, France
 {youakim.badr, frederique.laforest, andre.flory}@lisi.insa-lyon.fr*

Keywords: semi-structured data, user-written documents, information extraction, databases.

Abstract: Most database applications capture their data using graphical forms. Fields have limited size and predefined type. Although data in fields is associated with constraints, it is modeled to conform to a rigid schema. Unfortunately, too many constraints on data are not convenient as most human activities are document-centric. In fact, documents are a natural way for human information production and consumption. In this paper, we introduce DRUID, a document capturing and wrapping system. It ensures flexible and well-adapted information capture based on a Document User Interface and information retrieval based on databases. DRUID relies on a wrapper that transforms documents contents into relevant data. Also, it provides an expressive specification language for end-users to write domain-related extraction patterns. We validate our information system with a prototype.

1 Introduction

The most primitive approach to capture data is to fill fields in graphic forms provided by database applications. They contain fields with limited size and predefined data types. Often, these fields are associated with constraints to ensure that the stored data conforms to a rigid schema. DBMS efficiently manipulate data using standard query languages such as SQL. The scenario is not the same in document-centric applications, where data are semi-structured. Markup Languages such as XML can be used to present semi-structured data under the form of Data-Centric Documents (DCD) that contain fine grained data, as usually used in EDI. Many query languages, such as XQL (Robie, 1998)] manipulate DCDs. Another class of documents, called XML Paragraph-Centric Documents (PCD), is characterized by large grained fragments, usually tagged paragraphs of free text. It becomes an alternate and convivial approach to capture all kind of users data. Today, PCDs are widely exchanged and their data reused in a large range of tasks. They open issues for applications such as patient records, technical documents, business reports, newspapers... Although PCDs are

attractive for capturing structured text or natural free text, they require different types of processing philosophies. Users can render XSL style sheets and use browsers to display documents contents. This is not enough when the user needs advanced processing on documents contents, such as extract relevant data from text paragraphs, capture documents content and feed databases, do statistical study, decision support or mining. However, current Document Management Systems do not support this type of processing. On the other hand, a DBMS is convenient for such processing on data. The aim of our research is the conception of an Information System that ensures the flexibility of documents to capture paragraphs of free text by means of PCDs, and at the same time efficiency of databases to retrieve data. In this paper, we present DRUID and its implementation:

1. A Document User Interface to capture end-users data and generate PCDs.
2. A light wrapper extracts relevant information in paragraphs and return back the result as tuples.
3. A transformation processor builds XML DCDs. Thus, XML query languages can be applied.
4. A data mapping processor feeds a relational databases. Thus, SQL queries can be applied.
5. A high level specification language for users to design extraction patterns in regular expressions.

The remainder of the paper is organized as follows. Section 2 presents an overview of the system design. Section 3 describes the DRUID core module as well as its transformation processor, Xtractor and data mapping processor. In section 4 and 5 we present the implementation and we discuss related works.

2 Overall System Design

DRUID architecture has 3 main modules (Figure 1):

- Document User Interface

The DUI allows end users to write paragraphs of natural text. Available types of paragraphs depend on the document subject which is chosen by the user from a predefined list. In fact, it refers to a DTD that specifies paragraphs tags, as well as data that should be included in the header of the document. Once the user submits the document, an XML PCD is generated and forwarded to the DCM. The DUI also allows to query data using SQL on the Relational-DB or XML on the XML-DB. Answers include data in databases and links to captured PCDs.

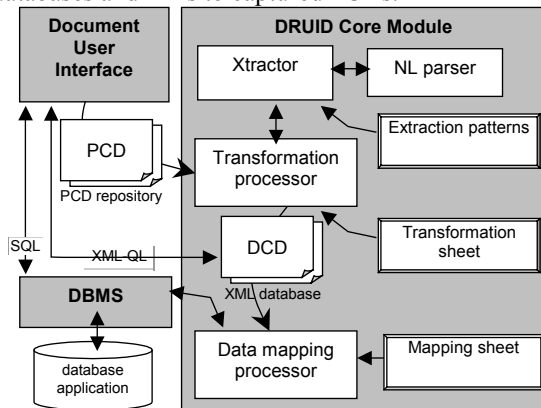


Figure 1: Overall DRUID logical architecture

DRUID Core Module

The DCM mainly consists of three sub-modules:

- The Transformation Processor aims at extracting relevant data from captured PCDs by invoking the Xtractor, it performs processing and restructuring as well as building XML-DB of DCDs containing only extracted data.
- The Data Mapping Processor, which creates tuples from the DCD content and inserts them into the relational database. It also provides a gateway to send DUI queries to DBMS.

- The Xtractor module extracts relevant data from paragraphs. It uses a Natural Language Parser for paragraphs processing and a specification language to write extraction patterns.

The transformation processor reads a PCD generated by the DUI and looks for its transformation sheet to execute its content. The transformation sheet contains different types of rules, each rule is dedicated to one step in the transformation. The processor may invoke the Xtractor module to manage extraction rules that effectively extract relevant information from paragraphs. The processor ends by generating a DCD, which contains only relevant data. Finally, the data mapping processor uses a mapping sheet that describes a relational schema to transform the DCD content into SQL instructions in order to feed the database.

- Database Management System

The DBMS manages the database and the PCDs repository. The database is a relational domain-related database (e.g. medical record). It is not a keywords index, but it is intended to store form-captured data as well as document-captured information. In implementation, we have inserted columns to store links (XLink) to PCDs that have been used to capture data. Queries on the database can return information in two forms: data generated by the Xtractor, and documents written by end-users. In the PCD repository, documents are entirely stored without fragmentation neither automatic indexing of their contents. We store the PCD repository in the database to take advantages of DBMS services. In the following sections, we limit our description to DCM and transformation processor. See (Badr et al., 2001, Laforest et al., 2002) for information on DUI and database.

3 Druid Core Module

The DCM does information extraction from free text and databases feeding. In other words, it allows the mapping between the PCD and a database. We create an intermediate DCD before feeding the database. In such a way, data processing can be applied and data restructuring can be used so that the intermediate DCD obeys the database schema and builds an equivalent XML database.

The DCM comprises two phases: the transformation phase, which transforms a PCD into a DCD, and the

mapping phase, which transfers the content of a DCD to the application database. As a consequence, we implemented two processors: the transformation processor and the data mapping processor.

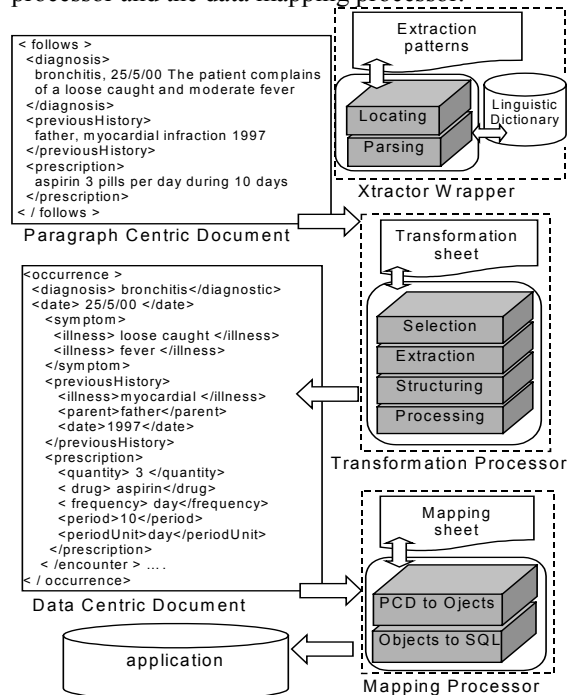


Figure 2: DRUID Core Module Architecture

3.1 Transformation processor

Data generation from a PCD begins with applying rules to select paragraphs, extract relevant information, process extracted data, and restructure it according to a database schema. The transformation processor reads a PCD and gets the transformation sheet associated with its DTD. This sheet contains a list of transform templates, similar to template-patterns in XSLT.

```

<Transform paragraph='XPath_Expression' >
  content
</Transform>
  
```

Each transformation template is defined by a *Transform* element. The *paragraph* attribute and its value specify the selection rule. *Content* has other rules whose execution results in the target DCD.

- **Selection rules**

A selection rule is a match pattern that identifies paragraphs by operating on the PCD logical structure. To write match patterns, we use a sub-set

of XPath. We define the selection rule syntax as an attribute-value pair of transformation template.

- **Extraction rules**

An extraction rule can be viewed as a function call that specifies arguments and returned values. The body of this function is defined internally as an extraction pattern. Extraction patterns are customized FSTs (Silberztein, 2000) that locate matching sequences of words by describing their relevant context. Within an extraction pattern, each matching sequence of words marks out a slot. An extraction pattern can return many slots (see section 3.2.2). An extraction rule has the syntax shown here:

```

<Extract pattern = 'Extraction_pattern_name' >
  <Slot name='slotname1' />
  .....
  <Slot name='slotnamen' />
</Extract >
where Extraction_pattern_name is a pointer to an entry in
the dictionary of extraction patterns.
  
```

- **Structuring rules**

Structuring rules are used to tag relevant data and insert it into the target document. They do operations like generating constant text, removing XML elements content, moving or reordering XML elements or duplicating text. Structuring rules are mainly conceived to put data in a format that is easy to feed the relational database. They present a superset of standard XSL elements (`xsl:element`, `xsl:attribute`, `xsl:copy`, `xsl:value-of`). To gather a combination of extraction rules into the target output within a single element, we have added the *construct* element type.

```

<Construct frame='tagname' type='frag | defrag'>
  content
</Construct>
where content refers either to extraction rules or to
processing rules.
  
```

Extracted information are tagged and copied to the target document. If the type attribute is “*defrag*”, the entire text is put into the target document.

- **Processing rules**

Processing rules apply operations on extracted information. The result is tagged and inserted into the target document. The basic categories of operations are: string operations (comparison, substitution...), arithmetic operations (+, -, /, *, etc.), mathematical functions (ceiling, floor, abs...). A processing rule has the following syntax:

```

<value-of operation = 'operation' />,
where the result of an operation is inserted in the target.
  
```

3.2 Xtractor

Here we illustrate the information extraction part of our system, the core specification language, and our paragraphs pre-processing mechanism to apply extraction patterns. This Xtractor module can be seen as a light wrapper, as it wraps a paragraph into a tuple that conforms to a model.

3.2.1 Paragraph pre-processing

Elements in a PCD are tagged paragraphs of natural language text. Tags add semantics and define a context for their contents. One motivation for document fragmentation is to construct small fragments of a mixture of grammatical or free text. Often, extracting domain-specific information from these short fragments is a limited and easier task than full text understanding.

Since DRUID is an application oriented system, it expresses its applications by natural language dictionaries, and by domain specific dictionaries. Users can refer to these dictionaries by meta symbols to write extraction patterns, such as `<disease>` to refer to any disease in the disease dictionary. In order to extract relevant information, Xtractor applies its dictionaries against PCD content and defines a scenario of text language pre-processing as follows: (i) validation ensures that the input is a text and it is clean of control characters, (ii) pre-processing identifies sentences in paragraphs, delimits unambiguous compound words, and marks off special tokens such as elided words, (iii) lexical analysis consists of applying language dictionaries and domain specific dictionaries, (iv) disambiguation removes ambiguity when a word corresponds to more than one entry in dictionaries. After the pre-processing stage, the result is a text vocabulary and indexed dictionaries. Various search operations can thus be done by applying extraction patterns. The output delimits relevant information.

3.2.2 User-defined extraction patterns

Regular expressions (RegExps) seem to be a good candidate for a pattern specification language but:

- One cannot define meta-words e.g. `<disease>` to denote any word in a disease dictionary,
- One cannot reuse a RegExp to build a new one,
- A long and nested RegExp becomes unreadable.

DRUID provides end users with a high level specification language to construct dictionaries of extraction patterns. It is RegExp-like with powerful add-in functionalities. DRUID supplies a compiler to check the syntax and to translate each extraction pattern into a Finite State Transducer (FST).

Figure 3(a) presents a complex extraction patterns, in other sense a RegExp of words over a text. It is not easy to maintain such a pattern. It becomes more readable if it is decomposed into layers (Figure 3(b)). We have defined four layers:

1. The *terms layer* is composed of a finite set of terms. A term is either (i) *A form* i.e. a sequence of letters delimited by separators e.g. whitespace etc. (ii) A meta-words to describe a number `<NB>`, a word `<MOT>`, an empty word `<E>`, or a reference to lemma in dictionaries.
2. The *expression layer* contains a finite set of expressions; we denote by an expression a concatenation of terms separated by a separator. We say that an expression holds if we find a sequence of words that matches the expression.
3. The *slots layer* is made of alternates of expressions, we say that a slot *holds* if one of its expressions holds. Unary operators could be applied on expressions in a slots such as kleene (*), optional (?), and one or more (+).
4. An *extraction pattern* over a paragraph is a finite and unordered set of slots. Conditions over slots may modify the possible combinations: unary operators over slots increase search paths and a restriction over slot sequence reduces possible combinations.

3.2.3 Extraction patterns implementation

Informally, an FST is a device that recognizes some sequences (e.g. of words or terms) in the input, and associate them with some outputs (e.g. linguistic information). From a different point of view, a FST expresses the context to locate a matching sequence. All possible combinations defined in an extraction pattern can be illustrated by a skeleton graph (Figure 3(d)) with one start node and one end node where every path represents one and only one possible combination of slots. The skeleton graph of an extraction pattern can be translated into a FST: each slot can be replaced by its equivalent elementary FST (Figure 3(c)). It is not tricky to find an elementary FST for expressions and then for each of their slots.

Eventually, the extraction pattern becomes a very huge FST and it is even hard to be generated using a graphical tool (see Figure 3).

Assuming that a FST can reproduce the matching sequence in the output, we can insert additional nodes within the FST to tag relevant sequences and

merge their outputs within the original text. That is why, we introduce *named term* as an identifier assigned to a term in order to add semantics and to delimit it by tags in the output. Formally, we note a named term as identifier[term].

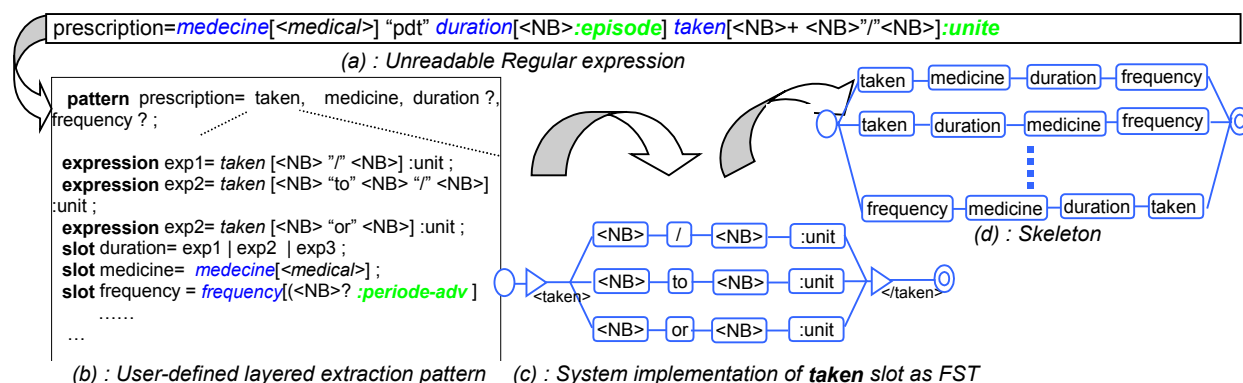


Figure 3: Extraction pattern for prescription paragraph and its equivalent FST

4 DRUID implementation

We developed a primary release of a DRUID prototype that supports capture, transformation, and data mapping. The prototype is written in java and is based on widely accepted XML standards.

The Document User Interface uses the Swing library, the EJB technology (Weblogic platform). XML operations use the Apache Xerces parser. The transformation Processor is built on top of the SAXON XSLT processor (Kay, 2002). It allows us to extend its API packages to implement our transformation rules as extension XML elements. The transformation processor employs JAXP1.1 to

process PCD and DCD during the transformation. Advanced documents parsing uses the Xerces parser. We take advantage of the well known INTEX (Silberztein, 1993) NLP to apply FSTs on paragraphs. In INTEX, texts and dictionaries are all represented by FSTs. Therefore, all the natural language operations that the Xtractor has to perform are translated into operations on FSTs in INTEX. We have developed a full FST compiler called eReL with JavaCC; it transforms each user extraction pattern into a FST. We use XML-DBMS (Bourret et al., 2000) for transferring data between DCD and relational databases.

5 Related works

DRUID concerns three major topics in the document domain: 1) capturing paragraphs of free text. 2) managing extracted data from documents through XML query languages and SQL. 3) user-defined extraction patterns. To our knowledge, an equivalent of the whole system has not been proposed by any other researches and hence we do not discuss it further in this section. Instead, we concentrate on the approaches taken for extracting and restructuring data in documents.

The trends that are most closely related to our own are designing wrappers. Many studies have been investigated to build wrappers for HTML pages (Crescenz et al., 2001), XML documents (Muslea, 1999) or for natural free text (Agichtein et al., 2000). Most wrappers, however, fail to extract properly relevant data, although documents appears to be structured in a highly regular fashion. One of the reasons is that data is semi-structured, it is not easy to find some uniform syntactic clues or even linguistic knowledge to extract attributes correctly.

Another trend is to query and manage data in data centric documents. It consists in mapping irregular data to a traditional database models. This type of mapping is supported by many systems such as Stored (Deutsch et al., 1999), Lorel (Abiteboul et al., 1997), UnQL (Buneman et al., 1996), struQL (Fernandez et al., 1998).

Currently there is an increasing use of paragraph centric documents. Few systems are conceived for PCDs; NoDoSE (Adelberg, 1998) provides a semi-automatic system to extract data from instances of a document type. It partially supports paragraph centric documents. Based on ontologies, (Embley et al., 1998) formulates rules to extract relevant information and apply a recognizer to produce values of tuples attributes in a generated database schema. This approach is limited to relatively small ontological models and small sets of keywords.

6 Conclusion

In this paper, we have presented DRUID, an information system which captures data through a document user interface and builds concurrently relational data and XML data. By coupling document

and database models, DRUID serves two purposes. First, it provides flexibility to produce and exchange documents. Second, it offers efficiency to manage their data by means of query languages.

DRUID architecture is based on XML technologies, finite state transducers and linguistic tools. Despite the diversity of components, the system is controlled by rules; the transformation sheets as well as by a specification language for extraction patterns.

Since the early beginning of the DRUID project, we applied the system to the medical domain. A real contribution of physicians led to build a small corpus of cardiology and pneumology PCDs. The results of using DRUID are promising. Currently, we are intending to enlarge the corpus and tuning extraction patterns to cover as much as possible the experiment domain. We are also working on empirical results to measure precision and recall ratios in order to compare our Xtractor with other state of the art wrappers. In addition, we are planning to generate automatically the transformation sheet skeletons, and to investigate in machine learning to offer an alternate approach for constructing extraction patterns. Moreover, the DUI is a subject of our attention to embed multimedia objects in respect with W3C recommendations.

References

- Abiteboul et al., 1997. *The lorel query language for semi-structured data*. Int. J. on Digital Libraries.
- Adelberg, 1998. *NoDoSE: a tool for semi-Automatically Extracing Structured and Semistructured Data from Text Documents*. ACM SIGMOD conf.: 283-94.
- Agichtein et al., 2000. *Combining Strategies for Extracting Relations from Text*. ACM SIGMOD Workshop on Data Mining and Knowledge Discovery
- Badr et al., 2001. *Transformation rules from semi-structured XML documents to database model*. AICSSA, Beirut, Lebanon.
- Bourret et al., 2000. *A Generic Load/Extract Utility for Data Transfer Between XML Documents and Relational Databases*. 2nd Int. Workshop on Advanced Issues of EC and Web-based Information Systems (WECWIS)

- Buneman et al., 1996. *A query language and optimization techniques for unstructured data (UnQL)*. In Proc. SIGMOD Conf.,
- Crescenz et al., 2001. *RoadRunner: Towards Automatic Data Extraction from Large Web Sites*, VLDB
- Deutsch et al., 1999. *Storing semistructured data with STORED*, In ACM SIGMOD Conf.
- Embley et al., 1998. *Ontology-Based Extraction and Structuring of Information from Data-Rich Unstructured Documents*. CIKM: 52-59
- Fernandez et al., 1998. *Catching the boat with Strudel: experiences with a web-site management system*. In Proc of ACM-SIGMOD Conf.
- Kay, 2002. SAXON XSLT Processor 7.2, <http://saxon.sourceforge.net/>
- Laforest et al., 2002. *Using Weakly Structured Documents at the User-Interface Level to Fill in a Classical Database*. Chapter in: *Advanced Topics in Database Research*, K. Siau (ed.) Idea Group Publishing
- Muslea, 1999. *Extraction patterns for information extraction tasks: A survey*. In AAAI-99 Workshop on machine learning for information extraction
- Robie et al., 1998. XML Query Language (XQL). <http://www.w3.org/TandS/QL/QL98/pp/xql.html>
- Silberztein, 1993. *Dictionnaires Electroniques et Analyse Lexicale du Français--Le Système INTEX*, Masson, Paris, France.
- Silberztein, 2000. *INTEX: an FST toolbox*. Theoretical Computer Science, (234)33-46.