A first approach to the automatic recognition of structural patterns in XML documents

Angelo Di Iorio, Silvio Peroni, Francesco Poggi, Fabio Vitali
Department of Computer Science
University of Bologna
Italy
{diiorio,essepuntato,fpoggi,fabio}@cs.unibo.it

ABSTRACT

XML is among the preferred formats for storing the structure of documents such as scientific articles, manuals, documentation, literary works, etc. Sometimes publishers adopt established and well-known vocabularies such as DocBook and TEI, other times they create partially or entirely new ones that better deal with the particular requirements of their documents. The (explicit and implicit) requirements of use in these vocabularies often follow well-established patterns, creating meta-structures (the block, the container, the inline element, etc.) that persist across vocabularies and authors and that describe a truer and more general conceptualization of the documents' building blocks. Addressing such meta-structures not only gives a better insight of what documents really are composed of, but provides abstract and more general mechanisms to work on documents regardless of the availability of specific schemas, tools and presentation stylesheets. In this paper we introduce a schema-independent theory based on eleven *structural patterns*. We provide a definition of such patterns and how they synthesize characteristics emerging from real markup documents. Additionally, we propose an algorithm that allows us to identify the pattern of each element in a set of homogeneous markup documents.

Categories and Subject Descriptors

I.7.2 [**Document And Text Processing**]: Document Preparation—*Markup languages*; I.7.2 [**Document And Text Processing**]: Document Capture— *Document analysis*

General Terms

Algorithms, Theory

Keywords

XML, descriptive markup, document visualisation, pattern recognition, structural patterns

1. INTRODUCTION

Publishing has been part of a revolution in the last twenty years. The proliferation of digital formats and tools for digital publishing is undeniable, and the same World Wide Web is constantly used for publishing content, even personal, even by inexpert users. XML still plays a crucial role in this evolving scenario: XML-encoded documents such as books, articles, legal acts, reports, specifications are ubiquitous, and XML-based languages are widely used as intermediate formats for conversion and communication, and as final formats for publication.

In many cases, communities use well-known vocabularies such as DocBook [15] and TEI [14], other times they use customized vocabularies tailored for their purposes, others they completely invent new languages. It is not surprising that in all these solutions designers have to deal with complex constraints within the language. They are required to cover heterogeneous situations and to foresee possible (validation) mistakes and misinterpretations by the final users. All these difficulties contribute to produce rich and complex schemas, that often require a lot of effort to be fully understood and applied.

This paper discusses an alternative approach for studying and designing XML documents and schemas. Instead of focusing on the compliance to schemas that capture **all** aspects of a domain - from those general and used in most of the documents up to the irregularities and specific cases that are actually useful in a few cases – we propose to shift the analysis to a **meta-level** and to study classes of elements, i.e. patterns, that persist across documents and that distil the conceptualization of the documents and their components.

We have been investigating patterns for XML documents for some time [3] [4]. In this paper, we give a full description of our theory about *structural patterns*, meta-structures that we consider sufficient to express what authors of documents and schemas mostly need. We also provide a formal definition of our patterns and their relations, and discuss how they synthesize characteristics emerging from real markup documents.

A pattern-based classification can be used for different purposes. For instance, it can be adopted as reference model to extract information from legacy documents: rather than trying to derive (a posteriori) constraints on those documents, users can look for those patterns and derive the structural role of each piece of information. Similarly, compliance to patterns can be checked retrospectively in order to verify how user really follow certain grammars, how they share design approaches, and to what extent their documents follow community guidelines.

The adoption of structural patterns is useful for creating new documents and schemas too. They reduce choices, thus reduce possibility of errors and misinterpretations. Yet, some specific cases and constraints are not covered by patterns but patterns still capture the most relevant information of documents. Our goal is to investigate which are the most important and shared features of well-engineered documents, which of them should (and can) be extracted, which can be neglected and under which circumstances.

Notice also that a simpler model eases documents' processing by future applications: even those applications not yet existing today will be more reliable and easy-to-build if dealing with simpler and less ambiguous structures. The reliability and re-usability of patterns make them good solutions even when creating new XML documents, besides bringing benefits to human readers in terms of readability and noise minimization.

In fact, our approach is minimalist. We believe that it is possible to derive from any schema a sub-schema that is fully pattern-based and that authors actually adhere to in most of their documents. Towards this goal, we present some preliminary studies that illustrate how a significative set of XML documents produced by different authors within a given community – DocBook documents written by the authors of the Balisage Conference Series – follows the same meta-structural organisation. The paper also introduces the algorithm used for the analysis, that is able to identify the pattern of each element in a set of homogeneous documents. We exploit such automatic patterns recognition for automatically generating reasonable CSS stylesheets for the visualisation of XML documents – or, better, for their pattern-based counterparts – without making any assumption on the implicit semantics defined by the vocabulary in consideration.

The rest of the paper is organised as follows. In Section 2 we give an overview of our theory of structural patterns, while details of our theory and a formal description are provided in Section 3. In Section 4 we illustrate our algorithm for the automatic recognition of structural patterns and its application for automatic visualization. In Section 5 we present the experiments on our algorithm and we discuss the outcome of these experiments. In Section 6 we discuss some relevant work about the recognition of structures and meta-structures in XML documents. Finally, we conclude in Section 7 presenting some development we plan for the near future.

2. STRUCTURAL PATTERNS

The idea of using patterns to produce reusable and high-quality assets is not new in the literature. Software engineers [5], architects and designers often use – or indeed *reuse* – patterns to handle problems that occur over and over again. Patterns have also been studied to modularize and customize web ontologies [12]. They guarantee the flexibility and maintainability of concepts and solutions in several heterogeneous scenarios.

We have been investigating patterns for XML documents (e.g., [3] [4]) to understand how the structure of digital documents can be segmented into atomic components, which can be addressed independently and manipulated for different purposes. Instead of defining a large number of complex and diversified structures, a small number of *structural patterns* needs to be found that are sufficient to express what most users need.

The two main characterizing aspects of such set of patterns should be:

- *orthogonality* – each pattern needs to have a specific goal and to fit a specific context. The orthogonality between patterns makes it possible to associate a single pattern to each of the most common situations in document design. Then, whenever a designer has a particular need he/she has to only select the corresponding pattern and to apply it;

- *assemblability* – each pattern can be used only in some locations (within other patterns). Although this may seem a limitation, such strictness improves the expressiveness and non-ambiguity of patterns. By limiting the possible choices, patterns prevent the creation of uncontrolled and misleading content structures. This characteristic still allows the presence of overlapping items – for example, a block that contains two different inlines that overlap upon the same segment continues to be a valid structure in terms of patterns because its content model is not violated, even though the presence of overlapping descendants.

These patterns allow authors to create unambiguous, manageable and well-structured documents. The regularity of pattern-based documents makes it possible to perform easily complex operations even when knowing very little about the documents' vocabulary. Designers can implement more reliable and efficient tools, can make hypothesis regarding the meanings of document fragments, can identify singularities and can study global properties of sets of documents.

In particular, the automatic recognition of structural patterns can help in the visualization of XML documents. In particular, given a set of schema-homogeneous XML documents, our idea is to identify markup elements and their related compliant structural patterns knowing neither the implicit semantics of markup elements nor any presentational stylesheet associate with the schema.

In Section 3 we derive the structural patterns using description logic formulas[1] to provide their formal definitions from the abstract conceptualization provided by the abstract patterns of Fig. 1.

3. A THEORY OF PATTERNS

In this section we introduce general properties of markup elements and organize them in abstract classes from which we derive the actual patterns we describe in our model. The overall structure of our pattern theory is summarized in Fig. 1. Nine abstract patterns (in yellow) are used to generate eleven instanceable patterns (in blue) that constitute the core of our model, as detailed in Table 1.

3.1 Basic properties of markup content models

Each structural pattern is characterised by general properties that restrict their elements to be compliant with a

[1]A brief introduction to description logic can be found in [10] and [8].

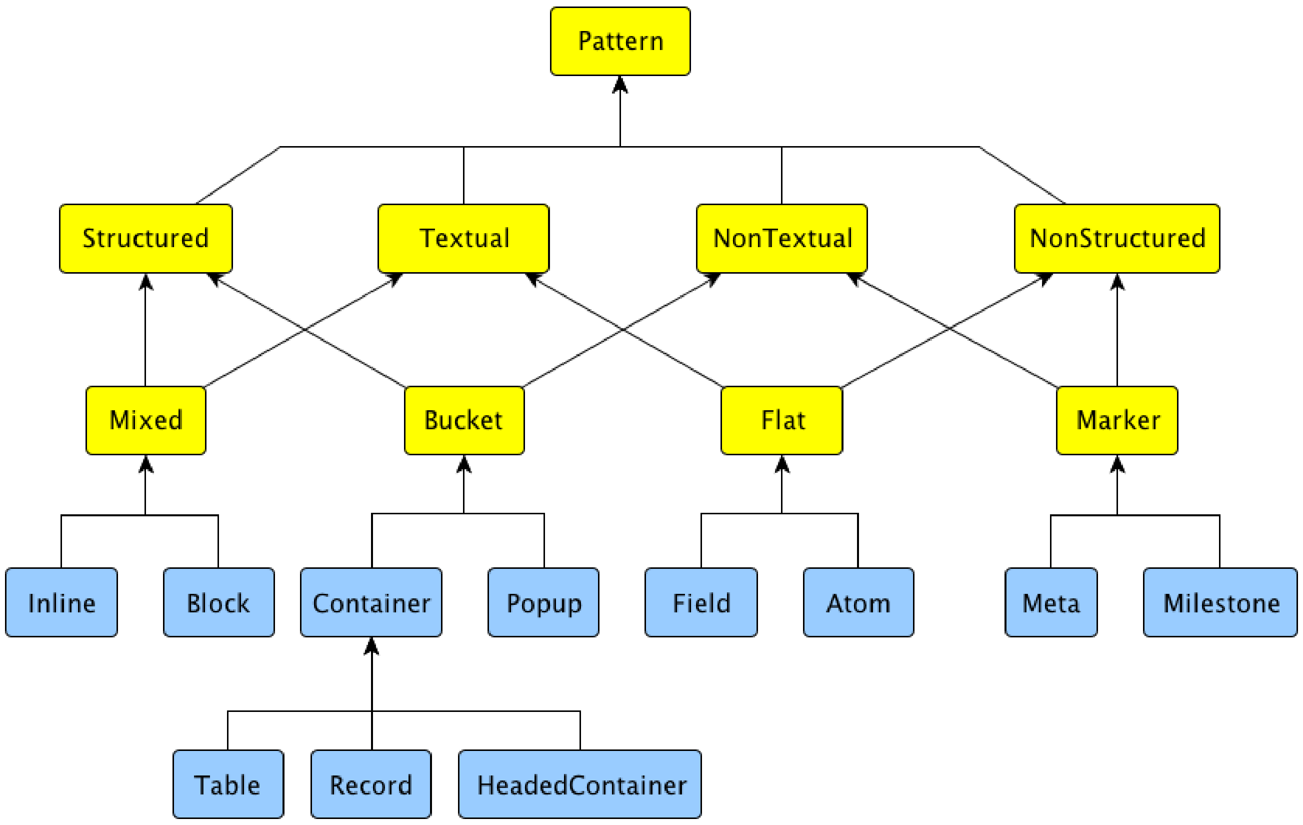

Figure 1: The classes defining the hierarchical structure of the structural patterns defined in our model. The arrows indicate sub-class relationships between patterns (e.g. Mixed is sub-class of Structured).

particular content model. Namely, we organize markup elements in four disjoint classes according to their ability to contain text and/or other elements.

We define *Textual* as the class of elements that *can have textual content* in their content models and *NonTextual* as the class of elements that cannot. These two classes are disjointed. We also define *Structured* as the class of elements that can *contain other elements*, and *NonStructured* as the class of elements that cannot. These two classes are disjoint.

```
Textual ⊑ ⊤
NonTextual ⊑ ⊤
NonTextual ≡ ¬ Textual
Textual ⊓ NonTextual ⊑ ⊥
Structured ⊑ ⊤
NonStructured ⊑ ⊤
NonStructured ≡ ¬ Structured
Structured ⊓ NonStructured ⊑ ⊥
```

We define the property *contains* (and its inverse *isContainedBy*) on *Structured* to indicate the markup elements its individuals contain:

```
∃contains.⊤ ⊑ Structured
isContainedBy ≡ contains-
```

While the content model of structured elements can contain any kind of optional and repeatable selection of elements, we need to be able to define some restrictions to their element repeatability. We thus define the boolean properties *canContainHomogeneousElements*, true if the element can contain elements that share the same name[2] , and *canContainHeterogeneousElements*, true if an element can contain elements with different names. In addition, we define *containsAsHeader* as a sub-property of *contains* to specify when a structured-based element contains *header* elements.

```
∃canContainHomogeneousElements.⊤ ⊑ Structured
⊤ ⊑ ≤1canContainHomogeneousElements
∃canContainHeterogeneousElements.⊤ ⊑
    Structured
⊤ ⊑ ≤1canContainHeterogeneousElements
containsAsHeader ⊑ contains
```

By combining the four classes defined above we are able to generate four new classes:

- class *Mixed*. Individuals of this class can contain other elements and text nodes;
- class *Bucket*. Individual of this class can contain other elements but no text nodes;
- class *Flat*. Individual of this class can contain text nodes but no elements;
- class *Marker*. Individual of this class can contain neither text nodes nor elements.

[2]By *name* we mean the pair *(namespace, general identifier)* of XML elements.

Table 1: Eleven structural patterns for descriptive documents.

Pattern	Description	Example (DocBook)
Atom	Any simple box of text, without internal substructures (simple content) that is allowed in a mixed content structure but not in a container.	email, code
Block	Any container of text and other substructures except for (even recursively) other block elements. The pattern is meant to represent block-level elements such as paragraphs.	para, caption
Container	Any container of a sequence of other substructures and that does not directly contain text. The pattern is meant to represent higher document structures that give shape and organization to a text document, but do not directly include the content of the document.	bibliography, preface
Field	Any simple box of text, without internal substructures (simple content) that is allowed in a container but not in a mixed content structure.	pubdate, publishername
Headed Container	Any container starting with a head of one or more block elements. The pattern is usually meant to represent nested hierarchical elements (such as sections, subsections, etc., as well as their headings).	section, chapter
Inline	Any container of text and other substructures, including (even recursively) other inline elements. The pattern is meant to represent inline-level styles such as bold, italic, etc.	emphasis
Meta	Any content-less structure (but data could be specified in attributes) that is allowed in a container but not in a mixed content structure. The pattern is meant to represent metadata elements that assert things about the document, but are disconnected to its actual text content.	imagedata, colspec
Milestone	Any content-less structure (but data could be specified in attributes) that is allowed in a mixed content structure but not in a container. The pattern is meant to represent locations within the text content that are relevant for any reason.	xref, co
Popup	Any structure that, while still not allowing text content inside itself, is nonetheless found in a mixed content context. The pattern is meant to represent complex substructures that interrupt but do not break the main flow of the text, such as footnotes.	footnote, tip
Record	Any container that does not allow substructures to repeat themselves internally. The pattern is meant to represent database records with their variety of (non-repeatable) fields.	address, revision
Table	Any container that allows a repetition of homogeneous substructures. The pattern is meant to represent a table of a database with its content of multiple similarly structured records.	tr, keywordset

These classes are defined as follows:

```
Mixed ⊑ Structured ⊓ Textual
Bucket ⊑ Structured ⊓ NonTextual
Flat ⊑ Textual ⊓ NonStructured
Marker ⊑ NonTextual ⊓ NonStructured
```

Given these eight classes, we can now define our structural patterns which will be described in the next section with examples from DocBook [15].

3.2 Structural patterns

We define the class *Block* to organise the document content as a sequence of nestable elements and text nodes. In particular, elements following this pattern can contain text and other elements compliant with the patterns *Inline*, *Atom*, *Milestones* and *Popup,* which will be introduced in the following. Furthermore, it is a requirement that block elements are contained only by *container* and *po pup* elements, as follows:

```
Block ≡
  Mixed ⊓
  ∀isContainedBy.(Container ⊔ Popup)
Block ⊑ ∀contains.(
  Inline ⊔ Atom ⊔ Milestone ⊔ Popup)
```

Examples of DocBook elements typically used as compliant with the *Block* pattern are *para* and *caption.*

We next define the class *Inline* with the same use and content model of the pattern *Block*, but differing primarily because:

- inline elements can contain other elements compliant with the same pattern while block elements cannot;
- inline elements must always be contained by other block or inline elements and by no other type of element.

These constraints also imply that inline elements cannot be used as root elements of documents and that the class *Block* is disjoint with the class *Inline* (i.e., a markup element cannot be a block and an inline at the same time), as specified by the following axioms:

```
Inline ≡
  Mixed ⊓
  ∀isContainedBy.(Inline ⊔ Block)
Inline ⊑
  ∀contains.(
    Inline ⊔ Atom ⊔ Milestone ⊔ Popup) ⊓
  ∃isContainedBy.(Inline ⊔ Block)
Block ⊓ Inline ⊑ ⊥
```

Examples of DocBook elements typically used as compliant with the *Inline* pattern are *link* and *emphasis*.

The class *Atom* is defined to describe literal text that is part of the document body. Moreover, similarly to *Inline*, elements following the *Atom* pattern can only be contained within block or inline elements (and consequently they also cannot be used as root elements of documents). It can contain textual content and no other elements, as shown in the following definition:

```
Atom ≡
  Flat ⊓
  ∀isContainedBy.(Inline ⊔ Block)
Atom ⊑ ∃isContainedBy.(Inline ⊔ Block)
```

Examples of DocBook elements typically used as compliant with the *Atom* pattern are *email* and *code*.

We next define the class *Milestone* to contain neither other elements nor textual content. Moreover, similarly to *Inline*, elements following the *Milestone* pattern can only be contained within block or inline elements (and consequently they also cannot be used as root elements of documents). The formal definition of this class is as follows:

```
Milestone ≡
  Marker ⊓
  ∀isContainedBy.(Inline ⊔ Block)
Milestone ⊑ ∃isContainedBy.(Inline ⊔ Block)
```

The distinctive characteristic of the pattern *Milestone* is the *location* it assumes within the document. Examples of DocBook elements typically used as compliant with the *Milestone* pattern are *xref* and *co*.

We next define the pattern *Popup* as that Bucket that is only present within block and inline elements. Moreover, similarly to *Inline*, elements following the *Popup* pattern can only be contained within block or inline elements (and consequently they also cannot be used as root elements of documents), as shown in the following excerpt:

```
Popup ≡
  Bucket ⊓
  ∀isContainedBy.(Inline ⊔ Block)
Popup ⊑
  ∀contains.(
    Container ⊔ Field ⊔ Meta ⊔ Block) ⊓
  ∃isContainedBy.(Inline ⊔ Block)
```

Popup elements are used whenever complex structures need to be placed within content elements such as paragraphs. Examples of DocBook elements typically used as compliant with the *Popup* pattern are *footnote* and *tip*.

We next define the class Meta also for elements that contain neither other elements nor textual content. Contrarily to the pattern *Milestone*, which was meant to describe markup elements that impact the document because of their location, the main feature of its disjoint sibling is the mere *existence*, independently from the position it has within the document. *Meta* elements convey metadata information about the document or part of it, independently of where they are (e.g., the elements *imagedata* or *colspec* in DocBook). Thus, meta elements can be contained only within container or popup elements, as follows:

```
Meta ≡
  Marker ⊓
  ∀isContainedBy.(Container ⊔ Popup)
Meta ⊓ Milestone ⊑ ⊥
```

The class *Field* is defined to describe literal metadata or text that is not really part of the document body, contrarily to its disjointed sibling *Atom*. Its main difference with *Meta* is that *Field* can contain textual content, as shown by the following definition:

```
Field ≡
  Flat ⊓
  ∀isContainedBy.(Container ⊔ Popup)
Field ⊓ Atom ⊑ ⊥
```

Examples of DocBook elements typically used as compliant with the *Field* pattern are *pubdate* and *publishername*.

The next class to be defined is the pattern *Container*, which concerns the structural organization of a document. Elements following this pattern contain no textual content and contain only elements compliant with the patterns: *Meta*, *Field*, *Block* and any subtype of *Container*. It is disjointed with the pattern *Popup*, although they share the same content model. Its formalisation is as follows:

```
Container ≡
  Bucket ⊓
  ∀isContainedBy.(Container ⊔ Popup)
Container ⊑
  ∀contains.(Container ⊔ Field ⊔ Meta ⊔ Block)
Container ⊓ Popup ⊑ ⊥
```

Examples of DocBook elements typically used as compliant with the *Container* pattern are *bibliography* and *preface*.

Container has a very general definition. It is thus useful to define subclasses that describe situations all ascribable to the Container pattern, but that have a typicality worth of their own pattern. The first such pattern is *Table*. Elements compliant with *Table* must contain only homogeneous elements (but they can be repeated), as follows:

```
Table ≡
  Container ⊓
  canContainHomogeneousElements:true ⊓
  canContainHeterogeneousElements:false
```

Representative DocBook elements that are commonly used as compliant with the pattern *Table* are *tr* and *keywordset*.

The pattern *Record* has the opposite restriction than *Table*: its element can only contain heterogeneous and non repeatable elements, as in the following axioms:

```
Record ≡
  Container ⊓
  canContainHomogeneousElements:false ⊓
  canContainHeterogeneousElements:true
```

Examples of DocBook elements typically used as compliant with the *Record* pattern are *address* and *revision*.

Next, the pattern *HeadedContainer* is the subclass of *Container* whose content model need to begin with one or more block elements (the heading), specified through the property *containsAsHeader* as follows:

```
HeadedContainer ⊑
  Container ⊓
  ∀containsAsHeader.Block
```

Examples of DocBook elements typically used as compliant with the *HeadedContainer* pattern are *section* and *chapter*.

Finally, it is also important to point out that all these subclasses of *Container* are disjoint, as follows:

```
Table ⊓ Record ⊑ ⊥
HeadedContainer ⊓ Record ⊑ ⊥
HeadedContainer ⊓ Table ⊑ ⊥
```

In the following section we introduce an algorithm for determining the pattern that best describes each element of an XML document.

4. RETRIEVING STRUCTURAL PATTERNS IN DOCUMENTS

In order to verify whether the theory of patterns presented in the previous section is adequate and complete, we describe here an algorithm we have created to assign one of the patterns to the elements of one or more XML documents using the same vocabulary, relying on no background information about the vocabulary, its intended meaning and its schema.

During the first preliminary step the algorithm performs a visit of the whole input document in order to group the elements with the same pair *(namespace,name)*[3]. This task of associating a structural pattern to each element in the input document is performed in four different steps that will be described in the following.

Step 1. After this introductory phase the algorithm considers each group and assigns their elements to one of the following categories Section 3.1:

1. *Mixed* if the elements contain both non-empty text nodes and other elements. It is sufficient that one element of the group contains both non-empty text and other elements to add all the elements of the group to this category.

2. *Marker* if the elements contain neither other elements nor non-empty text nodes. It is necessary that all the elements of the group are empty to add all the elements to this category.

3. *Textual* if the elements contain text nodes but no elements.

4. *Structured* if the elements contain only elements but no text nodes.

These categories correspond to the first four abstract classes defined in the pattern ontology. The algorithm then proceeds to identify more properties of the elements, thereby moving the elements to more specific and eventually concrete pattern classes.

[3]We refer to this pair as its *general identifier* (as a nostalgic remembrance of the good old SGML days). In the rest of this discussion we will refer to these groups of element as "the element with general identifier *id*", or just "the element *id*", and we will talk simply of "elements" referring to all the elements with the same general identifier.

Step 2. In this step the definitions of the patterns are used to derive rules that identify even more specific characteristics of the element groups. Consider for instance the first rule:

- If a *Mixed* element (or of one of its subclasses) contains a *Textual* element, then the *Textual* element becomes *Inline*.

In fact, given the definitions of the patterns Section 2, elements classified as *Textual* can only belong to concrete classes *Inline*, *Block* or *Mixed*, but among these the only pattern that can be contained by *Mixed* elements is *Inline*. Using similar arguments we can derive the following rules:

- If an *Inline* element is contained within a *Structured* element, then the *Structured* element becomes *Mixed*.

- If a *Marker* element is contained within a *Mixed* element (or any of its subclasses), then the *Marker* element becomes *Milestone*.

- If a *Structured* element is contained within a *Mixed* element (or any of its subclasses), then the *Structured* element becomes *Popup*.

These rules are tested sequentially over the whole document. Since the effects of any rule may change the state of the either of the container and contained element, the premises of another rule could become true therefore triggering its execution, which could trigger yet another rule, and so on, the algorithm tests these rules over all the element groups of the document until no rule can be executed any more.

Step 3. After these steps the algorithm moves the elements that are still associated to abstract patterns into concrete ones. For example, since *Mixed* is an abstract class and has two concrete subclasses *Inline* or *Block*, and since in the previous steps we have identified all the *Inline* elements, the remaining *Mixed* elements can become *Block*. Similarly, *Textual* elements become either *Fields*[4], *Marker* elements become *Meta*, and *Structured* elements become plain *Containers*.

Step 4. The following step is to provide for the identification of the three *Container* subclasses, *Table*, *Record*, and *HeadedContainer*. The algorithm uses their basic properties to discern the Container elements. In particular the algorithm introduces the following shift rules:

- If each element in a group contains one or more elements with the same general identifier, then it becomes *Table*.

- If each element in a group contains many differently named elements with different general identifiers, then it becomes a *Record*.

- If each element in a group contains an initial list of *Block* elements with the given sequence of general identifiers, followed by a list of one or more elements whose identifier is not contained in the head, then it becomes a *HeadedContainer*.

[4]We decided not to consider the pattern *Atom* in the algorithm for simplicity, since in our experiments it was always captured by the pattern *Inline*.

5. TESTING THE THEORY

We finally performed a preliminary evaluation of the quality of the algorithm for the automatic recognition of structural patterns. To do so, we selected an XML vocabulary (i.e. DocBook), manually assigned each element of its schema to one of our patterns according to our understanding of the semantics of the element (as illustrated in Table 2), and compared it to the outcome of the algorithm over a number of documents. Since the schema gives less strict definitions of its element than those allowed in our patterns, and since we compared a human analysis of the schema against an algorithmic evaluation of actual documents, we aimed to prove two important points of our theory: first of all, that there is a reasonable sub-schema of the main schema that is pattern-compliant, and, secondly, that in most cases authors can and will adhere to such subschema for their expressive needs even if the grammar of the language does not require them to do so. The full results of the test and the generic CSS produced from the patterns we recognised are available online at http://fpoggi.web.cs.unibo.it/DOCENG2012/patterns.html.

Table 2: The assignments of each element of the DocBook schema in consideration to one of our patterns.

Pattern	DocBook elements
Inline	biblioid, citation, code, emphasis, link, quote, subscript, superscript, email, jobtitle
Block	bibliomixed, mathphrase, orgname, para, subtitle, term, title, programlisting, td, th
Container	affiliation, author, equation, figure, info, informaltable, listitem, mediaobject, note, table, variablelist, blockquote, td, th
Table	abstract, caption, itemizedlist, keywordset, legalnotice, orderedlist, personblurb, tbody, thead, tr
Record	confgroup, imageobject, personname, varlistentry
Headed-Container	appendix, article, bibliography, section
Popup	footnote, blockquote, programlisting
Field	confdates, conftitle, firstname, keyword, othername, surname, email,jobtitle
Meta	col, imagedata
Milestone	xref

Specifically, we have chosen a particular subset of the DocBook language used by the Balisage Markup Conference[5], and 118 papers published in the years by that conference. This was chosen as a first experiment for several reasons: first of all, all the papers of the conference are freely available online[6], then, we know the community and the publication process and are personally certain that the authors of the papers are the actual authors of the XML versions available online (i.e., only a very limited editorial process affected the actual ML vocabulary chosen) and, finally, we know that the authors belong to a community composed of markup experts.

[5]http://balisage.net/tagset.html

[6]http://balisage.net/Proceedings/index.html

We first run the algorithm on each paper independently: the outcome of each single execution assigned a pattern to each element present in each input document, thereby creating a scheme of patterns for each paper. We then compared each execution against each other to create a single scheme of patterns for all documents. As shown under the heading "algorithm's outcomes" of the results available online, this phase created a clear polarization of the outcomes in 2 homogeneous groups, composed of 68 and 50 documents respectively, each with really similar pattern schemes to each other. By analysing the results, the group of 68 papers are a good match to the the assignments in Table 2 created manually on the schema, while the second group did not provide a good set of patterns for a number of reasons, the more evident of which is that the pattern *Inline* heavily prevails over the others, being assigned to more than a half of the elements.

Table 3: All the admissible shifts among patterns.

Patterns	Pattern shift
Block,Field	Block
Milestone, Inline	Inline
Meta, Field	Field
Meta, Block	Block

Even in the first group of papers we had problems. For instance, the algorithm was not able to assign one pattern to each element: thirty elements (i.e., 50% of the total) received the same pattern in all the 68 documents, while the others had two or more patterns assigned. From our point of view this variability should be interpreted neither as a mistake of the authors of those documents nor as wrong outcome of our algorithm. Rather, it means that different authors use the same element in different ways, such as happened to the element *note*. In this case, all the authors of the first set of papers used this element as a container for one or more blocks (e.g. paragraphs) – and the algorithm correctly associated the pattern *Container* (or one of its subclasses) to it. However, in five of these documents the element *note* did not contain repeated elements and thus the algorithm recognised it as *Record*, while in other six documents it contained homogeneous and repeated elements (and thus it was recognised as *Table*). In addition, in one document *note* contained always a sort of header (i.e. an element *title*), and consequently it was recognised as *HeadedContainer*.

In order to reduce the multiplicity of assignments, we applied three specific reduction strategies to all the element groups, so as to select the most suitable pattern in case of disagreement (such as the aforementioned multiple assignment to the element *note*):

1. First, we applied a *pattern shift*. If element *E* is associated to both pattern *P1* and *P2* and *P1* can be used in place of *P2*, then *E* has pattern *P1*. For instance, the element *para* has been recognised as both

Block (including both text and elements) and *Field* (including only text) by the algorithm. Since a *Block* element can happen to just contain text but Fields can never contain other elements, then *para* elements can be assigned to the pattern *Block* without problems. In Table 3 we illustrate all the possible pattern shifts.

2. We then applied a *discrimination rule for containers* , where we chose a specific kind of container whenever an element was associated to more that one subclass of containers. Namely, if an element ended up associated to two patterns (both subclasses of Container), then the element is associated to one of the subclasses if there was a clear disparity in the size of the two assignments (i.e., to the subclass that was selected more than 57% of the times), and to their superclass Container if no clear choice emerged (i.e., if both assignments ended up between 43% and 57% of the times).

3. Finally, we applied the *majority wins rule* to perform final discriminations in the remaining patterns, using a specific order for the comparisons and to solve equalities: first *Container* or any of its subclasses, then *Popup*, *Block*, *Inline*, *Milestone*, *Field*, and finally *Meta*. For instance, the element *orderedlist* was given pattern *Table* since it has been recognised twenty times as *Table* and only five times as *Popup*.

Table 4: The assignments returned by our process. The differences with the assignments illustrated in Table 2 are highlighted in italics.

Pattern	DocBook elements
Inline	biblioid, citation, code, emphasis, link, quote, subscript, superscript
Block	bibliomixed, mathphrase, orgname, para, subtitle, term, title, programlisting, td
Container	affiliation, *appendix*, figure, info, informaltable, mediaobject, note, table
Table	abstract, caption, *equation*, *imageobject*, itemizedlist, keywordset, legalnotice, orderedlist, personblurb, tbody, thead, tr, *variablelist*
Record	*author*, confgroup, *listitem*, personname, varlistentry
Headed-Container	article, bibliography, section
Popup	footnote, blockquote
Field	confdates, conftitle, firstname, keyword, othername, surname, email,jobtitle, *th*
Meta	col, imagedata
Milestone	xref

After disposing of all the ambiguous assignments, we obtained the list of pattern assignment shown in Table 4, where every element is associated with only one pattern. We then compared such assignments with the ones we illustrated in Table 2. Under the heading "Final discussion" of the results available at http://fpoggi.web.cs.unibo.it/DOCENG2012/ patterns.html it is possible to find a brief overview of all the stages of this process, together with the data related to the comparison with the assignments in Table 2. In addition, there are also links to download the code developed and to visualize these documents with the generic CSS produced from the recognised patterns.

In Table 5 we illustrate how many elements were assigned to the various patterns by our automatic process, highlighting also the amount of false-positive and false-negative assignments emerged from the comparison with the assignments in Table 2.

Table 5: The amount of elements assigned to each pattern by our automatic process, and the false positives and false negatives resulting from the comparison with the assignments in Table 2.

pattern	recognised	false positive	false negative
Inline	8	0	0
Block	9	0	1
Container	8	1	4
Table	12	3	0
Record	6	2	1
Headed-Container	3	0	1
Popup	2	0	0
Field	9	1	0
Meta	2	0	0
Milestone	1	0	0

The algorithm produced the same results as in Table 2 for 53 assignments over 60 (88%). We found the following issues on the other ones:

- some assignments (5 over 7) were *more specific.* In two ways: (i) the pattern assigned to an element was a subclass of the one we identified manually (for instance, the element *author* was recognized as *Record* instead of *Container*) or (ii) the recognized pattern could be shifted to the one identified manually (e.g., the element *th* has pattern *Field* instead of *Block*);

- one assignment detected a *more general* pattern. It is the case of the element *appendix* recognized as *Container* instead of *HeadedContainer*;

- one assignment was completely different from our interpretation, namely *imageobject* that was recognized as *Record* instead of *Table.* Notice that the patterns *Record* and *Table* are both (disjoint and not shiftable) subclasses of the same pattern *Container.*

Some "more specific" assignments may depend on this dataset. Some elements, in fact, are recognized as belonging to a class since the dataset does not contain examples of using that element in a different way. The algorithm does not found enough information to discriminate between patterns and to assign a more general characterization.

Overall, this preliminary results are very promising. In particular, they confirm that an incremental approach can be exploited to assign patterns to each element of a set of documents, even without knowing the schema that document is validated against. Moreover, experiments showed that the way authors actually use document elements, even if those elements do not follow patterns in the schema declaration, does not differ so much from their pattern-based counterpart.

Analysing in detail the remaining fifty documents we discarded in our experiments, we established that several elements assigned to the pattern *Inline* were the result of an ambiguous use (although allowed by the Balisage DTD) of particular elements. For instance, let us consider the element *figure*. Sometimes, some authors have been used that element within a paragraph (element *para*), thus implicitly assigning to it the role of *Popup*. Other times, *figure* has been used as direct child of *Containers* and, thus, it has been recognised as a proper *Container* by our algorithm (since a *Popup* cannot be contained by a *Container*). Of docurse, when these two conditions happen in the same document, the algorithmic computation tends to diverge to a "limit case", in which the root of the document is considered as a *Block* and all the other elements are *Inlines*. In order to avoid this issue, we are investigating strategies to calculate the most appropriate pattern scheme in such a scenario. In addition, we are investigating feasible approaches for the automatic suggestion of document changes (or *refactor operations*) so as to restructure these documents according to our pattern theory.

6. RELATED WORKS

In this section we provide an overview of some recent literature that is relevant to our work. For instance, Tannier et al. [13], starting from previous works by Lini et al. [11] and Colazzo et al. [2], describe an algorithm to assign each XML element in a document to one of three different categories: *hard tag*, *soft tag* and *jump tag*. *Hard tags* are all those elements that are commonly used to structure the document content in different blocks – they are the most frequent kind of elements and usually "interrupt the linearity of a text" and, in the DocBook vocabulary [15], correspond to, e.g., *para*, *section*, *table* etc. Next, all the elements that identify significant text fragments and are "transparent while reading the text" are *soft tags* – they are mostly inline elements with some presentation rule (e.g., in DocBook, *emphasis*, *link*, *xref*, etc. shown as bold, italic, in colour, etc.). Finally, the *jump tags* are elements that are logically "detached from the surrounding text" and that give access to related information – e.g., in DocBook, *footnote*, *comment*, *tip*, etc. Tannier et al. also introduce algorithms to assign XML elements to these categories by means of NLP tools. It is interesting to note that the "soft" category is very close to our *Inline* pattern, and that the group "hard" comprises *Block* and *Container* (and its subclasses), and "jump" includes our pattern *Popup*. This latter pattern, on the other hand, is a very good example of *reading context* as introduced by Tannier's work.

Zou et al. [16] categorise HTML elements as belonging to one of two classes: *inline* and *line-break* tags. Inline elements all those that do not introduce provide horizontal breaks in the visualisation of documents – e.g., *em*, *a*, *strong* and *q*, while line-break elements are those that do so – e.g., *p*, *div*, *ul*, *table* and *blockquote*. Based on this categorisation and a Hidden Markov Model the authors try to identify the structural role (e.g., title, author, affiliation, abstract, etc.) of textual fragments of medical journal articles expressed as HTML pages. Although this approach is tailored for HTML, and the algorithm exploits some features of the language, there are similarities with our work. In particular, the class *Inline* is very similar to ours. The class *line-break* is related to our idea of *Container*, that we further refined in subclasses (*Table*, *Record* and *headedContainer*)

The idea of distinguishing tags in two groups, those that do not interrupt the stream of flow and those that create nested structures, makes our approach also close to the work of Cardoso et al. [1]. The authors introduce an algorithm to segment news-oriented Web pages so as to recognise the title, publication date and the body of the story. They evaluate both the structure (the DOM hierarchy) and the presentation (the individual CSS styles) of a set of sample documents and train a machine-learning model to try to assess the role of HTML elements in other documents. Similar to us, Cardoso et al. point out that their approach is schema-independent and can thus be applied to other markup languages as well (although they tested only HTML documents).

Structural patterns have also been deeply studied by Koh et al. [9]. In particular, they identify text fragments and images that can work as surrogates of the whole document, where surrogates are "information elements selected from a specific document, which can be used in place of the original document". They address the issue of identifying *junk structures*, such as navigational elements of Web sites, advertisements, footers, etc., that usually do not carry the meaning of a document. Their approach is based on a pattern recognition algorithm that segments the XML elements of the document according to *tag patterns*, i.e., recurring hierarchies of nested elements that "contextualize the structured markup of text within a document". They find that junk structures are often described by similarly structured markup in different documents, and thus some tag patterns are crucial for their identification as junk within real HTML pages.

Finally it is worth mentioning Georg et al. [7], who introduce an NLP approach to the automatic processing of medical texts such as clinical guidelines, in order to identify linguistic patterns that support the identification of the markup structure of documents. This approach allowed the development of the system for the automatic visualisation and presentation of unstructured documents. In a more recent paper [6] Georg et al. illustrate an extension of such a work in which they introduce an improved version of their approach. This makes their work close to our idea of converting documents into pattern-based ones and using schema-independent visualization tools. Eventually this project exploits linguistic patterns - connected by precise containment and proximity rules - to create a pattern-based projection of the original document, on top of which they extract and visualize information through XSLT transformations.

7. CONCLUSIONS

The radical simplification of the documents (meta-)model is a key aspect of the theory presented in this paper: we believe that a small set of patterns is enough to express what users, authors and publishers need in most cases. In our vision, it is possible to find a reasonable pattern-based subschema from any schema and to use such a simplified version for validating most of the documents written by a community. The preliminary experiments on the Proceedings of Balisage Conference have confirmed such idea.

The application of automatic patterns' recognition and visualization to the DocBook galaxy is probably not so useful in practice, since a lot of powerful tools already exist for this language. The point is that the theory and algorithm discussed in this paper are schema-independent: patterns can be exploited to make sense of documents regardless of the availability of schemas, tools and presentation stylesheets. For instance, we plan to process the pattern-based conceptualization of documents to automatically generate table of contents (from *Containers* and *HeadedContainers*), to extract terms and glossaries (from *Inlines* and *Fields*), to extract structured data (from *Records* and *Tables*) and to automatically build editors and navigators.

In the future, we plan to test our theory taking into account documents written in other XML formats, such as TEI, and by alternative communities of document authors. For example, it will be interesting to use structural patterns for retrospective analysis, in order to investigate how the authors/communities actually use certain grammars, to what extent they share design rules, to what extent they use specific constructs and validation rules. Similarly, we plan to investigate automatic mechanisms to restrict existing grammars in order to be patterns-based and to isolate non-pattern-based examples in large document sets.

8. REFERENCES

[1] Cardoso, E., Jabour, I., Laber, E., Rodrigues, R., Cardoso, P. (2011). An efficient language-independent method to extract content from news webpages. In Proceedings of the 2011 ACM symposium on Document engineering (DocEng11). DOI: 10.1145/2034691.2034720.

[2] Colazzo, D., Sartiani, C., Albano, A., Manghi, P., Ghelli, G., Lini, L., Paoli, M. (2002). A typed text retrieval query language for XML documents. In Journal of the American Society for Information Science and Technology, 53 (6): 467-488. DOI: 10.1002/asi.10059.

[3] Dattolo, A., Di Iorio, A., Duca, S., Feliziani, A.A., Vitali, F. (2007). Structural patterns for descriptive documents. In Baresi, L., Fraternali, P., Houben, G. (Eds.), Proceedings of the 7th International Conference on Web Engineering 2007 (ICWE 2007). DOI: 10.1007/978-3-540-73597-7_35.

[4] Di Iorio, A., Gubellini, D., Vitali, F. (2005). Design patterns for document substructures. In Proceedings of the Extreme Markup Languages 2005. Rockville, MD, USA: Mulberry Technologies, Inc. http://conferences.idealliance.org/extreme/html/2005/Vitali01/EML2005Vitali01.html (last visited June 29, 2012).

[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Boston, Massachusetts, USA: Addison-Wesley. ISBN: 0201633610.

[6] Georg, G., Hernault, H., Cavazza, M., Prendinger, H., Ishizuka, M. (2009). From Rhetorical Structures to Document Structure: Shallow Pragmatic Analysis for Document Engineering. In Proceedings of the 2009 ACM symposium on Document engineering (DocEng09). DOI: 10.1145/1600193.1600235.

[7] Georg, G., Jaulent, M. (2007). A Document Engineering Environment for Clinical Guidelines. In Proceeding of the 2007 ACM symposium on Document engineering (DocEng07). DOI: 10.1145/1284420.1284440.

[8] Horrocks, I., Patel-Schneider, P. F., McGuinness, D. L., Welty, C. A. (2007). OWL: A Description Logic Based Ontology Language for the Semantic Web. In Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., Patel-Schneider, P. F. (Eds.), The Description Logic Handbook: Theory, Implementation and Applications (2nd edition): 458-486. Cambridge, UK: Cambridge University Press. ISBN: 9780521876254.

[9] Koh, E., Caruso, D., Kerne, A., Gutierrez-Osuna, R. (2007). Elimination of junk document surrogate candidates through pattern recognition. In Proceedings of the 2007 ACM symposium on Document engineering (DocEng07). DOI: 10.1145/1284420.1284466.

[10] Krotzsch, M., Simancik, F., Horrocks, I. (2011). A Description Logic Primer. Ithaca, New York, New York: Cornell University Library. http://arxiv.org/pdf/1201.4089v1 (last visited June 29, 2012).

[11] Lini, L., Lombardini, D., Paoli, M., Colazzo, D., Sartiani, C. (2001). XTReSy: A Text Retrieval System for XML documents. In Augmenting Comprehension: Digital Tools for the History of Ideas.

[12] Presutti, V., Gangemi, A. (2008). Content Ontology Design Patterns as practical building blocks for web ontologies. In Li, Q., Spaccapietra, S., Yu, E. S. K., Olivé, A. (Eds.), Proceedings of the 27th International Conference on Conceptual Modeling (ER 2008). DOI: 10.1007/978-3-540-87877-3_11.

[13] Tannier, X., Girardot, J.,Mathieu, M. (2005). Classifying XML tags through "reading contexts". In Proceedings of the 2005 ACM symposium on Document engineering (DocEng05). DOI: 10.1145/1096601.1096638.

[14] Text Encoding Initiative Consortium (2005). TEI P5: Guidelines for Electronic Text Encoding and Interchange. Charlottesville, Virginia, USA: TEI Consortium. http://www.tei-c.org/Guidelines/P5 (last visited June 29, 2012).

[15] Walsh, N. (2010). DocBook 5: The Definitive Guide. Sebastopol, CA, USA: O'Really Media. Version 1.0.3. ISBN: 0596805029.

[16] Zou, J., Le, D., Thoma, G. R. (2007). Structure and Content Analysis for HTML Medical Articles: A Hidden Markov Model Approach. In Proceedings of the 2007 ACM symposium on Document engineering (DocEng07). DOI: 10.1145/1284420.1284468.