SgmlQL + XGQL = Powerful XML Pattern-Matching and Data-Manipulation in a Single Language

Jacques Le Maitre[(1)], Yves Marcoux[(2)] & Elisabeth Murisasco[(1)]

[(1)] SIS-Informatique
Université de Toulon et du Var
B.P. 132
F-83957 La Garde Cedex, France
{lemaitre, murisasco}@univ-tln.fr

[(2)] <GRDS> – EBSI
Université de Montréal
C.P. 6128, Succ. Centre-ville, Montréal, QC
Canada H3C 3J7
Yves.Marcoux@UMontreal.CA

Abstract

The presence of XML in many recent hypermedia management tools and methods (W3I3, SMIL, etc.) shows better than ever that both structural and textual criteria will continue to play a fundamental role in content-based multimedia retrieval and management. Extracting precise and fine-grained information from structured documents requires powerful pattern-matching. Most existing XML query languages are fairly limited in this respect. Powerful pattern-matching languages do exist, but usually lack the data manipulation capabilities required of a true information management environment. In this paper, we investigate how XGQL, a powerful pattern-matching language based on generalized grammars, can be integrated into SgmlQL, a complete XML retrieval and manipulation language.

1 Introduction

Multimedia and hypermedia systems are increasingly popular every day. As the body of available applications and corpora grows, though, the problems associated with managing, accessing, retrieving, sharing, and reusing the information elements contained in these applications and corpora become more and more important. Since reuse typically involves different readings of the same information by different people at different times and in different contexts, efficient management methods must allow hypermedia components to be attached to different access and/or categorization schemes, that may be user-dependent, and that may not even exist at content-creation time. In order to facilitate such "reattachment" of information components, a representation as perennial and neutral as possible must be adopted in the repository. Such representations often and typically involve some sort of textual information with a predictable structure, a case in point being metadata, various flavors of which have been a major focus of interest for the Internet community in recent years; see for example (Dublin Core 2000, GILS, 1999).

It is thus not too surprising that many recent hypermedia management tools and methods are actually based on structured-text representations. For instance, both W3I3 (Ceri, 1999) and SMIL (W3C, 1998b, 1999) are based on XML (eXtensible Markup Language) (W3C, 1998a), the current major structured-text standard, developed by W3C. This, together with the fact that text remains an important content-type in many hypermedia applications—especially documentary ones—, entails that both structural and textual-content criteria will continue to play a fundamental role in content-based multimedia retrieval and management.

The purpose of this paper is to propose a new approach to hypermedia information retrieval supporting both structural and textual-content criteria. While it is clear that the resulting language can query structured-text databases, it should also be noted that the language is able to express powerful structure-only queries, that could thus be applied to arbitrary multimedia databases, even

without any textual content. The language also has a link-traversal operator that allows directly querying hypermedia databases (such as intranet- and extranet-webs, as well as *the* Web).

Information retrieval in structured text has been extensively studied, so we need to explain how our approach differs form earlier proposals. Early efforts towards structured-text retrieval include extensions to traditional models, such as the relational model, various IR models, and object-oriented databases. A brief survey of these early efforts can be found in (Navarro et al., 1995), where the authors also discuss why these approaches do not satisfactorily address the problem of querying structured-text databases with both structural and textual-content criteria.

Novel approaches usually impose some sort of "flexible structure" to the database, most often a hierarchical one, and allow the formulation of mixed structural and textual-content criteria. A brief survey of such approaches is also given in (Navarro et al., 1995). Following (Gonnet et al., 1987), the kind of structure imposed on the database is usually a variant of context-free grammars (CFGs) (Hopcroft et al., 1999), which is quite natural for structured text. Modern flavors of these include SGML and XML document-type definitions (DTDs) and schemas (W3C, 2000b). One of the least restrictive model for database structure is given by the well-formedness constraints of XML. Incidentally, it is the model underlying both SgmlQL (Harie et al. 1996) and XGQL (Marcoux et al., 1997).

When it comes to the actual retrieval operations, the various proposed models vary greatly. Usually, there is a pattern language, more or less directly related to CFGs, supplemented by various mechanisms such as variable-binding, pattern-combination and data-manipulation operations, and even embedding in a full-fledged programming language. Of course, the purely combinatorial expressiveness of the pattern language has a significant impact on the expressiveness and flexibility of the resulting retrieval language.

One of the most expressive pattern language is that of *p-strings*, introduced in (Gonnet et al., 1987). Several other pattern languages have been proposed, among which tree-matching (Kilpeläinen et al., 1992, 1993), two-dimensional templates (Kuikka et al., 1997), PAT expressions (Salminin et al., 1992), and others (Navarro et al., 1995). In the conclusion of (Kilpeläinen et al., 1993), the authors mention that a "serious" pattern-matching language should include a negation operation, which would allow search criteria defined in terms of *not matching* a given pattern. In (Marcoux et al., 1997), a very powerful pattern language was introduced, based on generalized grammars, which subsumes CFGs in expressiveness, and adds various operations, *including* negation and conjunction. Because of these operations, the pattern language is strictly more powerful than CFGs. To our knowledge, this is the most expressive pattern language ever proposed as a basis for structured-text retrieval. The query language defined in (Marcoux et al., 1997) is hereafter referred to as XGQL (for XML Generalized-grammar Query Language).[1]

Another track of development was foreseen (and maybe actually triggered) by Gonnet and Tompa, when they mentioned in the conclusion of (Gonnet et al., 1987) that the functional approach was likely to be a suitable foundation for a semantic model of p-strings. Perhaps the most comprehensive achievement in this line is SgmlQL, a complete functional language integrating some of the p-string pattern-matching capabilities in the form of OQL-like queries.

As noted in (Navarro et al., 1995) (and as can be expected intuitively), there is usually a tradeoff between the expressiveness of the pattern language and the complexity of query evaluation. Not surprisingly, then, the query evaluation algorithm of XGQL has a fairly high (though polynomial-time) complexity, which makes it prohibitive in some settings. On the other hand, SgmlQL has a linear query evaluation algorithm, but lacks the full expressiveness of p-strings, let alone that of generalized grammars. There are (intuitively) simple queries that are not expressible in SgmlQL, because they rely intrinsically on pattern-matching capabilities that exceed those of the language.

Our proposal is to integrate the powerful pattern-matching capabilities of XGQL into SgmlQL, in order to increase its expressiveness. A side-effect is to provide XGQL with full search-results post-processing capabilities, which it totally lacks. The resulting language is to our knowledge the most powerful structured-text retrieval and processing language proposed so far in the literature. Our

---

[1] XML™ is a product of W3C®; XGQL is an independent proposal by Yves Marcoux.

proposed integration allows extracting small- to medium-sized portions of the database, and feeding them one at a time into an XGQL engine for local, fine-grained pattern-matching. Thus, the complexity of the XGQL pattern-matching can be limited, and need not relate to the size of the full database. The interest of this approach is supported, at least intuitively, by the fact that using powerful pattern-matching seems most natural in "local" settings, where fairly small portions of the database are thoroughly scrutinized for complex patterns. Macroscopic, higher-level manipulations and/or selections are not as naturally expressed as pattern-matching.

The remainder of this paper is organized as follows. The next two sections are respectively dedicated to overviews of XGQL and SgmlQL. Section 4 outlines how the proposed integration of the two languages can be defined. Section 5 looks at how the resulting language relates to other existing proposals. Finally, a short conclusion follows.

2 XGQL

2.1 Intuitive overview

As mentioned earlier, a document-base is a well-formed XML document instance. To define XGQL queries, we start with the formalism of CFGs, which we slightly modify so that it generates only strings (document instances) satisfying the XML well-formedness constraints. We add patterns, based on normal regular expressions, for matching textual content (#PCDATA) and generic IDs. Then, we add the SGML DTD constructors "|", "*", "+", and "?", as well as negation "!" and conjunction "^" operators. The OR "|" constructor is not really new to CFGs, being often used as a shorthand in writing CFGs. The XML sequence operator "," is not syntactically included; instead, we represent it implicitly by juxtaposition, as is customary in CFGs.

Some basic patterns can be used to match textual content and generic IDs. These are essentially normal regular expressions. Such basic patterns can be combined, using the above constructors, to define (possibly recursively) more complex ones. For example, given any two patterns α and β, the pattern <α>β</> will match any segment of the form <*id*>*text*</*id*>, where *id* matches α and *text* matches β. The pattern $\alpha \mid \beta$ will match any segment matching either α or β; $\alpha \wedge \beta$ will match any segment matching both α and β; $!\alpha$ will match any (well-formed) segment not matching α. Note that the choices of formalisms for matching textual content and generic IDs are entirely orthogonal to the rest of the model. In (Marcoux et al., 1997), these were also based on generalized grammars.

Note that the SGML unordered sequence operator "&", which we exclude, is also not present in XML. Our reason for excluding it is that it would not increase expressiveness, yet, as noted indirectly in (Kilpeläinen et al., 1993), it would lead to a complexity blow-up in query-evaluation. Moreover, most uses of "&" can be replaced by conjunctions without a blow-up in query size. For example, A & B & C is equivalent to the conjunction of the following 4 conditions: being a sequence of exactly three elements; containing A anywhere; containing B anywhere; and containing C anywhere.

Constructors "*", "+", and "?" are well-know not to increase the expressive power of CFGs. The negation operator "!" does increase expressiveness, allowing for example the definition of the non-context-free language $\{ww \mid w \in \{\texttt{"<A></A>"}, \texttt{"<B></B>"}\}^*\}$. By de Morgan's law, negation with OR automatically gives conjunction. Conjunction by itself (without negation) also increases expressiveness, allowing for example the definition of the non-context-free language $\{ \texttt{"<A></A>"}^n \texttt{"<B></B>"}^n \texttt{"<C></C>"}^n \mid n \geq 0 \}$. Care must be taken when adding negation to grammars because, since patterns are defined recursively, it could lead to inconsistencies, as in $A \rightarrow !A$. But there turns out to be a simple restriction that guarantees consistency.

A query is simply a generalized grammar of the kind just described. It might seem natural to imagine a query as being a pattern searched for in the document-base, and returning as results all segments matching it. However, there would be two drawbacks to such an approach. First, it would prevent us from writing a query returning only part of the searched pattern, for example: look for *A* followed by *B*, but return only *A*. Second, it would prevent us from expressing global context criteria in the grammar itself, for example: look for *A*, but only in the first sub-element of the document-base. Therefore, we define query-matching at the document-base level. Thus, in order

to return results, a query has to match the whole document-base; if it does, then it can also return portions of the document-base as results, based on indications included in the query itself. If we are in fact interested by *all* occurrences of some pattern *A* in the document base, it is straightforward to transform *A* into a query that matches the whole document-base, and returns as results the portions of the document-base matching *A*.

The indications as to which parts of the document-base should be returned as results of the query, are given by marking some sub-patterns of the grammar with the underscore character "_". For example, if we were searching for *A* followed by *B*, but wanted to return *A* only, we would write: *_A B*. The segments of the document-base that match marked sub-patterns are combined, based on the notion of *parse-tree*, to determine the overall result of the query. Apart from this mechanism, there is no way to process or manipulate the content of the document-base. This can be explained by the fact that XGQL was originally designed for purely navigational uses (Sévigny et al., 1996). In contrast, the integration with SgmlQL allows arbitrarily sophisticated manipulations of XGQL query results.

The theoretical background pertaining to generalized grammars will be the object of a separate paper, as will the detailed query-evaluation algorithm of XGQL. In (Marcoux et al., 1997), XGQL is presented using a data model that covers acyclic hypertext as well as XML well-formed documents; here, we adopt the simpler data model consisting of only XML well-formed documents. Thus, a document-base is any well-formed XML document instance. For a more thorough discussion of XGQL, the reader is referred to (Marcoux et al., 1997).

2.2 Queries

We define an *XGQL-query* as a tuple $Q=(V, G, T, S, P)$, where V is a finite non-empty set of *non-terminals* (sometimes called *variables*), G is a finite non-empty set of *genID characters*, T is a finite non-empty set of *text characters* (sometimes called *terminals*), $S \in V$ is a distinguished non-terminal called the *start-symbol*, and P is a set of at most $|V|$ *productions*, each of the form $A \rightarrow \alpha$, where $A \in V$ is called the *left-hand side* of the production, and α, the *right-hand side* of the production, is a *Q-expression*, as defined hereafter. For each $A \in V$, there is at most one production in P with a left-hand side equal to A; this production, if it exists, is called the *A*-production. Globally, P must satisfy a morphological condition, called the *no-RSMN* (no recursively self-matching negation) condition, which will be discussed later. Without loss of generality, we assume in the formal discussion that T is a subset of the Unicode character repertoire minus { "<", ">", "[", "]" }, and that G is a subset of T – { "/" }.

The notion of *Q-expression*, for a given query Q, is defined recursively as follows (the definition uses the auxiliary notions of *genID-expressions* and of *text-expressions*, which are only partly defined for the moment):

1. For all $w \in G^+$, w is a genID-expression.
2. For all $w \in T^*$, $[w]$ is a text-expression.
3. Any text-expression is a *Q*-expression.
4. For each $A \in V$, A is a *Q*-expression.
5. If α and β are *Q*-expressions, then, so are $\_\alpha$, $(\alpha?)$, $(\alpha*)$, $(\alpha+)$, $(!\alpha)$, $(\alpha\ \beta)$, $(\alpha \mid \beta)$, and $(\alpha \wedge \beta)$.
6. If γ is a genID-expression and α is a *Q*-expression, then $<\gamma>\alpha</>$ is a *Q*-expression.

GenID-expressions are patterns that can match generic identifiers within XML documents, and text-expressions are patterns that can match text (#PCDATA) segments. The forms defined above simply match "themselves" within XML documents; for instance, [] matches the empty string (""). We shall extend these definitions later, when we introduce a concrete syntax. *Q*-expressions, on their side, are patterns that can match well-formed segments within XML documents. We often omit parentheses in *Q*-expressions when it causes no ambiguity.

2.3 Matching

Before we assign meaning to *Q*-expressions, we must make sure that our introduction of negations does not lead to inconsistencies. We argue that the appropriate framework for this discussion is set-theoretic. In this setting, each production $A \rightarrow \alpha$ is viewed as the set-theoretic equation $A = \alpha$, and the whole set of productions *P* is viewed as a system of set-theoretic equations. The operations represented by juxtaposition, !, |, ^, *, +, and ? are respectively concatenation of sets of strings, set-complementation, set-union, set-intersection, Kleene-star, positive Kleene-star, and union with {""} (the empty-string singleton). The <γ>α</> construction represents the following operation on the sets of strings γ and α:

$$\cup_{w \in \gamma} (\{\text{"<"}\ w\ \text{">"}\}\ \alpha\ \{\text{"</"}\ w\ \text{">"}\}).$$

Note that complementation is with respect to the set of well-formed XML fragments. The underscore character "_" has no special meaning at this time, so it can be considered to represent the identity function.

It is easy to show that, if we restrict ourselves to the usual CFG constructors (or, equivalently, if we use neither ! nor ^), then the usual CFG derivation operation applied to XGQL queries always generates (as it does in CFGs) *simultaneous minimal* solutions for all variables in the query. For example, derivation applied to $A \rightarrow (A\ A) \mid$ <K>[]</> | [] generates {"<K></K>"}*, which is the minimal of an infinite number of values of *A* satisfying the equation. However, when we add complementation (!), then simultaneous minimal solutions for all variables do not always exist; in fact, the system of equations can even be inconsistent. For instance, the production $A \rightarrow (!A)$ represents the equation $A = \text{comp}(A)$, and is readily seen not to have any solution. The system of equations $A \rightarrow (!B)$ and $B \rightarrow (!A)$ has infinitely many solutions, but no simultaneous minimal solutions for both *A* and *B*. Note that being inconsistent (i.e., not having any solution) is quite different from having the empty set as the minimal solution: $A \rightarrow A$ has the empty set as its minimal solution, whereas $A \rightarrow (!A)$ has no solution at all.

Nevertheless, there turns out to be a simple, linear-time verifiable, morphological restriction on a set of productions that guarantees it to be consistent. This restriction, which is the *no recursively self-matching negation* (no-RSMN) condition mentioned earlier, is intuitively that there be no "pathological" recursion of the style $A \rightarrow (!A)$. Moreover, it is possible to define a notion of *joint-minimality* (slightly different from *simultaneous* minimality) on the solutions to a set of productions, in such a way that there always exists a unique jointly-minimal solution whenever the set of productions satisfies the no-RSMN condition. Thus, any XGQL query is guaranteed to possess a unique jointly-minimal solution. For each *Q*-expression α, we define the *language of* α, denoted $L(\alpha)$, as the set of strings obtained by interpreting the constructors in α as the operations described above, and assigning to the non-terminals in α their set-theoretic value in the unique jointly-minimal solution to the set of productions of *Q*. We say that a string σ *matches* α (or, inversely, that α *matches* a string σ) iff $\sigma \in L(\alpha)$.

2.4 Result of a query

To define the result of a query, we introduce the notions of *segment* and *parse-tree*. Let Δ be a well-formed XML document instance. We denote by $\|\Delta\|$ the length of Δ, as a string of Unicode characters. A *segment* of Δ is a pair of integers $\sigma = (x, y)$ such that $x \le y$, $x \ge 0$, and $y \le \|\Delta\|$. The *text* of σ, denoted $\Delta(\sigma)$, is the string of characters going from position $x+1$ to position y in Δ. Note that $\Delta = \Delta(0, \|\Delta\|)$. We say σ is *empty* iff $\Delta(\sigma) =$ "", that is, iff $x = y$. The *concatenation* of a segment $\sigma = (x, y)$ to a segment $\sigma' = (x', y')$, denoted $\sigma\|\sigma'$, is undefined if $y \neq x'$, or else defined as the segment (x, y').

We say a segment $\sigma = (x, y)$ is *well-formed* iff it is empty or, else, the three following conditions are met: (1) $\Delta(\sigma)$, as an XML document fragment, is well-formed; (2) if $\Delta(\sigma)$ starts with a character other than "<", then $x > 0$ and character position x in Δ is ">"; (3) if $\Delta(\sigma)$ ends with a character other than ">", then $y < \|\Delta\|$ and character position $y+1$ in Δ is "<". The sole purpose of this additional constraint on well-formedness of segments (as compared to well-formedness of

strings) is to make sure that text-expressions can only match whole #PCDATA segments (or empty segments).

Let χ be a *Q*-expression, Δ a well-formed XML document instance, and σ a well-formed segment of Δ. A χ-*parse-tree for* σ is a finite ordered rooted tree Π with the following properties:

1. Each vertex v of Π has two labels: an associated expression, denoted $\varepsilon(v)$, and an associated segment, denoted $\sigma(v)$. For all v, $\varepsilon(v)$ is a *Q*-expression and $\sigma(v)$ is a well-formed segment of Δ.
2. If r is the root of Π; then, $\varepsilon(r) = \chi$ and $\sigma(r) = \sigma$.
3. For all vertex v of Π, if $\varepsilon(v)$ is of the form $\_\alpha$, then v has a unique child with associated expression α and associated segment $\sigma(v)$.
4. For all vertex v of Π, if $\varepsilon(v)$ is of the form $(\alpha?)$, then either $\sigma(v)$ is empty and v has no child, or v has a unique child with associated expression α and associated segment $\sigma(v)$.
5. For all vertex v of Π, if $\varepsilon(v)$ is of the form (α^*), then either $\sigma(v)$ is empty and v has no child, or v has two children v_1 and v_2 such that $\varepsilon(v_1) = \alpha$, $\varepsilon(v_2) = (\alpha^*)$, and $\sigma(v) = \sigma(v_1)||\sigma(v_2)$.
6. For all vertex v of Π, if $\varepsilon(v)$ is of the form $(\alpha+)$, then either v has a unique child v_1 such that $\varepsilon(v_1) = \alpha$ and $\sigma(v) = \sigma(v_1)$, or v has two children v_1 and v_2 such that $\varepsilon(v_1) = \alpha$, $\varepsilon(v_2) = (\alpha+)$, and $\sigma(v) = \sigma(v1)||\sigma(v2)$.
7. For all vertex v of Π, if $\varepsilon(v)$ is of the form $(!\alpha)$, then v has no child and $\Delta(\sigma(v)) \notin L(\alpha)$.
8. For all vertex v of Π, if $\varepsilon(v)$ is of the form $(\alpha\ \beta)$, then v has two children v_1 and v_2 such that $\varepsilon(v_1) = \alpha$, $\varepsilon(v_2) = \beta$, and $\sigma(v) = \sigma(v_1)||\sigma(v_2)$.
9. For all vertex v of Π, if $\varepsilon(v)$ is of the form $(\alpha \mid \beta)$, then v has a unique child v_1 such that $\varepsilon(v_1) = \alpha$ or $\varepsilon(v_1) = \beta$, and $\sigma(v) = \sigma(v_1)$.
10. For all vertex v of Π, if $\varepsilon(v)$ is of the form $(\alpha \wedge \beta)$, then v has two children v_1 and v_2 such that $\varepsilon(v_1) = \alpha$, $\varepsilon(v_2) = \beta$, and $\sigma(v) = \sigma(v_1) = \sigma(v_2)$.
11. For all vertex v of Π, if $\varepsilon(v)=A$ for some $A \in V$, then v has a unique child v_1 such that $\varepsilon(v_1)$ is the right-hand side of the A-production and $\sigma(v) = \sigma(v_1)$.
12. For all vertex v of Π, if $\varepsilon(v)$ is a text-expression, then v has no child and $\Delta(\sigma(v))$ matches $\varepsilon(v)$.
13. For all vertex v of Π, if $\varepsilon(v)$ is of the form $<\gamma>\alpha</>$, then v has a unique child v_1 for which $\varepsilon(v_1) = \alpha$, and $\sigma(v) = (x, y)$ and $\sigma(v_1) = (x_1, y_1)$ are such that $x_1 > x+1$, $y_1 < y-1$, $\Delta(x+1, x_1-1)$ matches γ, and $\Delta(\sigma(v)) = (\text{"<"}\ \Delta(x+1, x_1-1)\ \text{">"}\ \Delta(\sigma(v_1))\ \text{"</"}\ \Delta(x+1, x_1-1)\ \text{">"})$.

Iff $\sigma = (0, ||\Delta||)$ and $\chi = S$, we also say that Π is a *Q-parse-tree for* Δ.

It can be shown that, for all well-formed segment σ and all *Q*-expression χ, there exists a χ-parse-tree for σ iff $\Delta(\sigma) \in L(\chi)$. Thus, our notion of parse-tree is similar to that of CFGs.

We say a vertex of Π is *marked* iff its associated expression is of the form $\_\alpha$. The *result* of Π, denoted $\rho(\Pi)$, is the ordered (possibly empty) list of (possibly empty) segments $\sigma(v_1)$, $\sigma(v_2)$, ..., $\sigma(v_n)$, where v_1, v_2, ..., v_n is the ordered list of marked vertices encountered during a preorder traversal of Π.

The *compiled form* of a well-formed XML document instance Δ, denoted $\kappa(\Delta)$, is obtained from Δ as follows: (1) all ignorable white-space and comments are removed; (2) all entity-references and marked-sections are resolved; (3) all attribute specifications are transformed into subelements of the form `<@attrib-name>normalized-attrib-value</@attrib-name>`. The "@" character being illegal in XML genIDs, markup introduced here cannot conflict with the original document markup. The compiled form of documents allows uniform treatment of elements and attributes, and constitutes a useful "canonical" representation of documents.

Finally, we say that a query *Q matches* a well-formed XML document instance Δ iff there exists a *Q*-parse-tree for κ(Δ). The *result* of *Q on* Δ, denoted *Q*(Δ), is defined as:

$$\cup_{\Pi \text{ is a } Q\text{-parse-tree for } \kappa(\Delta)} \{ \rho(\Pi) \}.$$

Thus, $Q(\Delta) \neq \emptyset$ iff *Q* matches Δ.

2.5 Concrete syntax

We introduce a concrete syntax for XGQL queries, and extend the forms of genID- and text-expressions.

The sets *G* of genID characters and *T* of text characters are always as in XML. For all $w \in G^+$, *w*, `!`*w*, *w*`%`, and `!`*w*`%` are genID-expressions, which respectively match: the string *w*, any string of G^+ except *w*, any string of G^+ that starts with *w*, and any string of G^+ that does not start with *w*; `%` is a genID-expression that matches all of G^+. For all $w \in T^+$ comprising only letters, digits, and spaces, `[`*w*`]`, `[!`*w*`]`, `[`*w*`%]`, and `[!`*w*`%]`, are text-expressions, which respectively match: all strings of T^* that contain *w* as a phrase, all strings of T^* that do not contain *w* as a phrase, all strings of T^* that contain a phrase that starts with *w*, and all strings of T^* that do not contain a phrase that starts with *w*; `[%]` is a text-expression that matches all of T^*, and `[]` is a text-expression that matches only the empty string. As has been pointed out earlier, the languages of genID- and text-expressions are entirely orthogonal to the rest of the model; thus, the languages presented here are mere illustrations of the possibilities.

The meaning of constructors `|`, `*`, `+`, `?`, `!`, `^`, `<.>.</>`, as well as juxtaposition and parentheses, is as presented above. As earlier, we often omit parentheses when it leads to no ambiguity; juxtaposition, `^`, and `|` are assumed to associate left.

An XGQL query is specified by writing its productions using the concrete syntax. All non-terminals (except `%`) are denoted by identifiers of the form `#name`. The productions are written as a sequence of *production specifications*, each of the form "`#name =` *expression* `;`". Non-terminal `%` cannot be specified in the query, and is always assumed to be specified as:

```
% = %* | <%>%</> | [%] ;
```

Thus, `%` will match any well-formed segment whatsoever. The start-symbol is always assumed to be non-terminal `#S`. The part "`#name =`" can be omitted from at most one production specification, in which case "`#S =`" is assumed.

If no production is specified for non-terminal `#S`, then it is assumed to be specified as:

```
#S = % (#R | <%>#S</>) % ;
```

In this case, `#R` can in effect be considered to be a pattern that is searched for in the whole document-base, regardless of the context in which it occurs. A colon "`:`" at the beginning of a production specification is interpreted as "`#R =`"

GenID-expressions of the form $w \in G^+$ are allowed outside the usual construct `<.>.</>`; they are then an abbreviation for `<`*w*`>%</>`. The "`;`" at the end of the last production specification can be omitted.

2.6 Example

Let Δ = κ("`<D><E T='a'>summer</E><G>summit</G></D>`"). Here is what Δ looks like:

```
<D><E><@T>a</@T>summer</E><G>summit</G></D>
0         1         2         3         4
01234567890123456789012345678901234567890123456789012
```

Character positions are given in origin 0 for easy locating of segments. For example, Δ(26, 39) = "`<G>summit</G>`". Note that κ(Δ) = Δ, because Δ is already in compiled form.

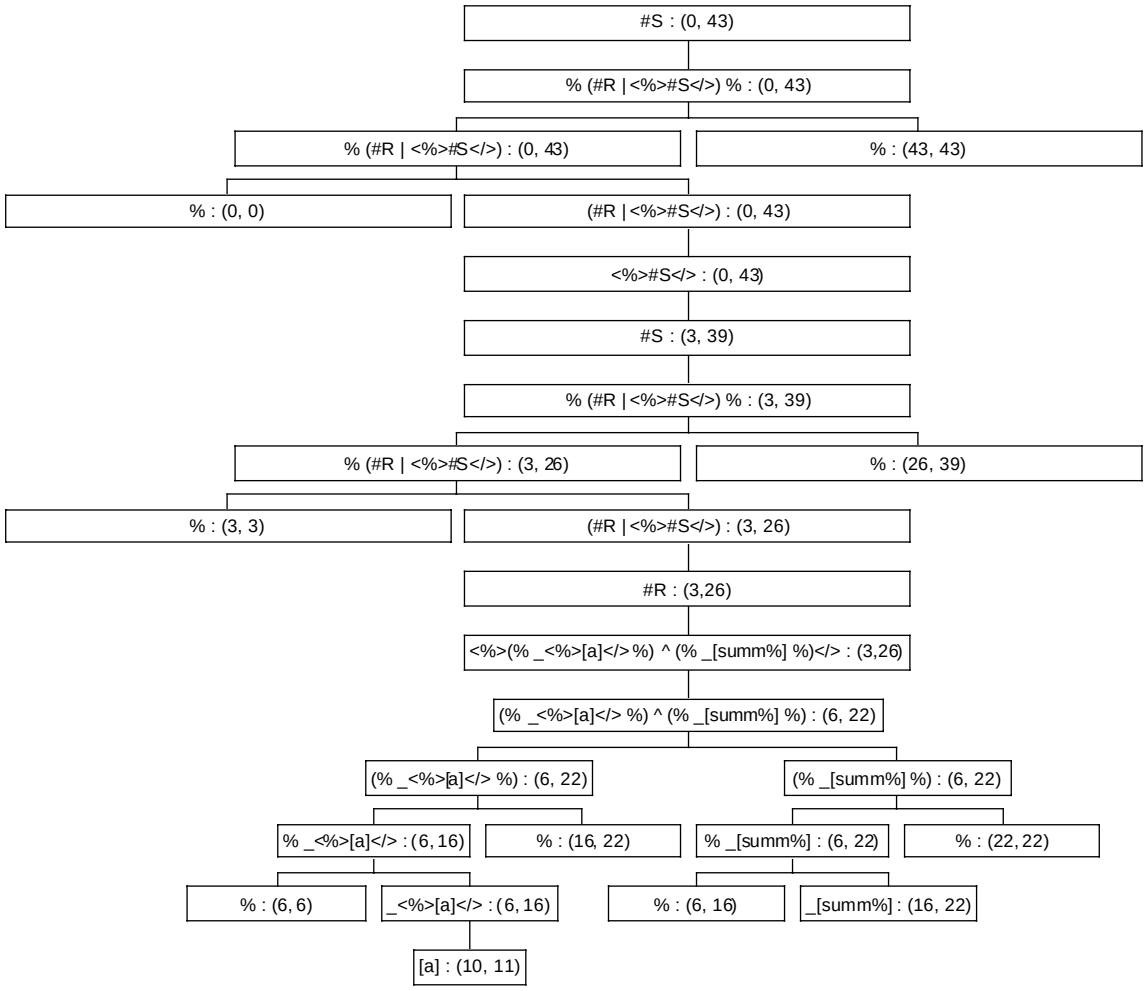
Now let query *Q* be specified by:

```
: <%>(% _<%>[a]</> %) ^ (% _[summ%] %)</> ;
```

Recall that the colon ":" means "#R =", and that #S has a default specification, which we have not overriden.

This query retrieves all #PCDATA sections containing a word starting with summ and having a sibling element containing the word a. Both the #PCDATA section and the sibling element in question are returned in the results. We use the notation $[(x_1, y_1) (x_2, y_2) ...]$ to denote lists of segments. We claim that $Q(\Delta) = \{[(6, 16) (16, 22)]\}$. There is only one result because there is only one #S-parse-tree for Δ. A representation of that parse-tree is shown in Figure 1 (we do not expand vertices that have % as an associated expression).

Figure 1 - Example parse-tree



The segments associated to the marked vertices encountered during a preorder traversal of the parse-tree are (6, 16) and (16, 22), in that order; hence, the result of the query. The reader might want to try and build other parse-trees for the given document and query, and see that there is no way to build a parse-tree other than the one shown.

2.7 Implementation issues

Although the semantics of XGQL involves parse-trees, it is not necessary to build them explicitly for query execution. The dynamic-programming algorithm of Cocke-Younger-Kasami for CFGs can be extended to support the new constructors of XGQL, as well as the generation of results. The query execution algorithm runs in $O(n^3)$, where n is the maximum number of direct children of any element in the document, which is often surprisingly low in practice, even for large documents. If we consider the query as part of the input, the run time increases by a factor of $O(m)$, where m is the length of the query, with an additive component of $O(m^2)$ for query preprocessing ($O(m^3)$ for optimized queries; see next paragraph). To this, we must add a time

linear in the length of the results to write them out. Result construction is lazy, and based on symbolic computation over an algebra of sets of segment lists. Only the actual results of the query are generated, after query evaluation is completed.

A large class of queries (including all the examples given below) can be optimized through NFA simulation, and are executable in time linear in the size of the document-base. A complete prototype in Java, built on top of James Clark's XP parser, and including query optimization, has been developed and is available from the second author. In practice, the performance of the prototype is usually surprisingly good, even with unoptimized queries and fairly large document-bases.

The task of writing XGQL queries is, like XML DTD creation, very much amenable to visual interfaces. Some visual elements in support of this task have been studied in (Sévigny et al. 1997).

3 SgmlQL

As its name indicates, SgmlQL is an SQL like language aimed to manipulate SGML and now XML documents.

3.1 Semantics

An XML document is a tree structured set of elements. An element is either atomic or compound. An atomic element is a character string (`#PCDATA` in XML). A compound element is described by its type name, by its attribute set and by its content which is an element list.

An element has a tree structure that we call its associated tree. The associated tree of an atomic element is reduced to this element. The associated tree of a compound element has a root which is its type name and a list of subtrees which are the associated trees of its subelements. Therefore the content structure of an element is an ordered forest.

SgmlQL has a functional semantics. A SgmlQL expression is either a constant (a type name, an attribute name, a character string or a number), a variable name, or an operation $op(e_1, \ldots, e_n)$ where *op* is an operator name and $e_1, \ldots, e_n$ are expressions. Type names or attributes names are in capital letters and variable names are prefixed by `$` (`$p` for example).

3.2 Main operators

The operator set of SgmlQL is very complete : about fifty operators. In this paper we only present those which are specific to the manipulation of XML elements and which make it possible: to build an element, to extract some components of an element, or to transform an element by replacing or deleting some of its subelements.

Extraction or transformation operations need to localize the affected components. Two location modes are taking in account:

- indexation to access type name, attributes, subelements or subjacent text of an element;
- filtering which consists in giving some information about the type or the content of the searched components without giving the absolute path to reach them.

Almost examples of this section are related to the manipulation of a bibliographic document bound to the `$doc` variable and having the following DTD where declarations of elements having a `#PCDATA` content are omitted:

```
<!ELEMENT BIB (ENTRY+)>
<!ELEMENT ENTRY (AUT+, TIT, CONF, YEAR, EDITOR?, FPAGE, LPAGE)>
<!ELEMENT AUT (NAME, SURNAME+)>
```

3.2.1 Construction of an element

Construction of an element is realized by the following operation:

`element` *n* [`attr:` *A*] `content:` *L*

where *n* is a type name, *A* is an attribute set and *L* is an element or an element list. Construction of an element list is realized by the following operation:

[e_1, ..., e_n]

where each e_i is an element. Construction of an attribute set is realized by the following operation:

{n_1= v_1, ..., n_k= v_k}

where n_1, …, n_k are attribute names and v_1, …, v_k are attribute values. For example:

(Q1)
```
element AUT
content: [(element NAME content: "Ullman"),
          (element SURNAME content "Jeffrey")]
```

Query Q1 constructs the following element:

```
<AUT><NAME>Ullman</NAME><SURNAME>Jeffrey</SURNAME></AUT>
```

3.2.2 Indexation

Indexation of an element *e* is realized by the following operations:

- `gi(`*e*`)`, the value of which is the type of *e*,
- `content(`*e*`)`, the value of which is the component list of *e*,
- `attr(`*e*`)`, the value of which is the attribute set of *e*,
- `text(`*e*`)`, the value of which is the left to right concatenation of the leaves of the associated tree of *e*.

Indexation of an element list *L* is realized by the following operations where *i* and *j* are positive integers denoting ranks in *L*:

- *L*[*i*], the value of which is the *i*th element of *L*,
- *L*[*i*:*j*], the value of which is the sublist of *L* between *i* and *j*.

Indexation of an attribute set *A* is realized by the following operations where *n*, n_1, …, n_k are attributes names in *A*:

- *a*`->`*n*, the value of which is the value of attribute *n* in *A*,
- *a* `restrict_to` {n_1, …, n_k}, the value of which is the subset of attributes of *A*, the names of which are in the given set.

For writing easiness, the `->` operator can be directly applied to an element. Expression *e*`->`*n* is equivalent to `attr(`*e*`)->`*n*.

3.2.3 Filtering

SgmlQL is equipped with a very powerful filtering operator : the `within` operator, which selects subelements of an element given a set of constraints on their names, on their hierachical level, on their content or on their attributes.

A filtering operation has the following general form:

F [`as` *var*] `within` *e* [`where` *p*]

where *e* is the element to be filtered ; *F* is a filter having the following form:

(`first` | `every` | `top` | `bottom`) [*niv*] [*N*]

niv is a constraint on the level of the searched subelements in the associated tree of *e*; *N* is a constraint on the type name of the searched elements, *var* is a variable bound to a subelement of *e*; *p* is a constraint (a boolean expression) on the content or on the attribute set of the *var*

subelement. Prefixed keywords `first`, `every`, `top` and `bottom` put a constraint on the number of searched subelements or on their depth level if the searched subelements are recursive (for example sections of a book).

The value of such filtering operation is an element list resulting from the following 5-steps process:

Step 1. The associated tree of *e* is scanned in post-order and each subelement *x* is passed to step 2.

Step 2. Subelement *x* is submitted to the level constraint *niv* which can have one of the following forms:

- *i*, where *i* is a level number: *x* is passed on to step 3 only if its level is *i*.
- [*i*:*j*], where *i* and *j* are level numbers: *x* is passed on to step 3 only if its level is between *i* and *j*.

(levels are numbered from 0 from root to leaves)

Step 3. Subelement *x* is submitted to the name constraint *N* which can have one of the following forms:

- *n*, where *n* is a type name : *x* is passed on to step 4 only if its type name is *n*.
- {n_1, …, n_k}, where each n_i is a type name : *x* is passed on to step 4 only if its type name is in the given set.
- `excluding` {n_1, …, n_k}, where each n_i is a type name : *x* is passed on to step 4 only if its type name is not in the given set.

Step 4. Subelement *x* is submitted to the prefix constraint which has one of the following form:

- `first` : *x* is passed on to step 5 only if it is the first element of *e*,
- `every` : *x* is passed on to step 5,
- `top` : *x* is passed on to step 5 only if it has no ancestor of type *n*,
- `bottom` : *x* is passed on to step 5 only if it has no successor of type *n*.

Step 5. Subelement *x* is submitted to the *p* constraint. For that, it is bound to *var*. If the value of *p* is true, *x* is definitively filtered.

Let us consider, for example, an XML document (`$my_book`) describing a book divided into nested sections (element `SEC`). Then query Q2 selects the most specific sections and query Q3 selects all sections, the text of which contains the substring "XML":

(Q2)
```
every bottom SEC within $my_book
```

(Q3)
```
every SEC as $s within $my_book where content($s) = "XML"
```

3.2.4 Selection

Selection is realized by the well known `select...from...where` operator. We describe it only through the two following examples:

(Q4)
```
select [first TIT within $e, every AUT within $e]
from $e in every ENTRY from $doc
where text(first YEAR within $e) = "1999"
```

(Q5)
```
select element BIBENTRY
       attr: YEAR = text(first YEAR within $e)
       content: [first TIT within $e,
                 element AUTHOR
                 content: [text((every SURNAME within $a)[1]),
                           text(first NAME within $a)] ]
from $e in every ENTRY within file "bibliography.xml",
     $a in every AUT within $e
where text(first CONF within $e) match "France"
```

```
and ((text(first LPAGE within $e) -
      text(first FPAGE within $e) + 1) > 5)
```

Query Q4 selects the name and the first surname of each author who published a paper in a conference held in 1999. Query Q5 selects papers having more than five pages and published in a conference held in France. For each reference, the name and the first surname of each author, the title of the paper, the name of the conference and the year are given. Let us notice that the year becomes an attribute, and that the name and surname of each author are concatenated in a new element AUTHOR.

3.2.5 Element transformation

To modify some subelements of an element, it is fastidious to use the `select…from…where` operator because that implies a complete reconstruction of this element. It is why SgmlQL is equipped with two specific operators `replace` and `remove` which make it possible to modify an element by replacing or removing some of its subelements.

The replacement operation has the following form:

`replace` *F* `as` *var* `within` *e* `by` *v*[`where` *p*]

where *F* is a filter as defined above (cf. 3.2.3), *var* is a variable, *e* is an expression, the value of which is an element, *p* is a boolean expression, *v* is an expression, the value of which is an element or an element list. This operation returns a new version *e'* of *e* in which subelements filtered by *F* and verifying the *p* condition have been replaced by the *v* element. The `where` clause is optional.

The remove operation has the following form:

`remove` *F* `as` *var* `within` *e* `where` *p*

where *F*, *var* and *p* are defined as in the `replace` operation. This operation returns a new version *e'* of *e* in which subelements filtered by *F* and verifying the *p* condition have been removed.

By example, query Q6 replace in each entry, the content of the AUT element by the first author followed by the expression "et al.", if the number of authors exceeds one:

(Q6)
```
replace every ENTRY as $e within file "bibliography.xml"
where count(every AUT within $e) >1)
by replace every AUT within $e
   by element AUT
      content: [first AUT within $e,
                element AL content:"et al.",
                remove every AUT within $e)]
```

3.2.6 Navigation

SgmlQL enables to manipulate linked documents through HREF attributes (Bruno et al., 1999). A specific operator is used to express and control hypertextual navigation. A navigation expression is composed of a start point (e.g an URL) and a path to follow over the hypertext. Path filters are defined as regular expressions. For example, `(->,_).(->,_)` is a path filter of length 2 where `->` and `_` are two elementary filters which respectively filter any link and any node. Path filters can be restricted according to the structure of the path, the followed links and the traversed documents. A path is seen as a list of pairs (*link*, *node*). The *link* component catches all the data needed to define the relation between the source and the target documents. This component is represented as an SGML element, and provides attributes as BASE and HREF, which refer to the source and the target documents, and LABEL, which labels the link. The node component is built from the document which is viewed like a node of the graph. The root tag is enriched with some attributes as URL which identifies the document or mimetype.

In order to produce the set of paths matching a pattern, a specific operator called `navigate` is defined. Variables can be bound to any component of a path filter. These variables are used to define constraints on the links and/or on the content of the nodes traversed. They can also be used to extract some components of these links and nodes. By example:

(Q7)
```
select first TITLE within $d
from navigate
node (http://www.navigo.com/paris/hist/musee-orsay.html)
     (-> as $l, _ as $d)
where $l->label match "impressionistic"
```

(Q8)
```
select element ITEM
       attr: URL = $d->URL
       content: text(first TITLE within $d)
from $node about "painting",
     navigate $node (-> as $l , _ as $d)
where $l->HREF match ".fr",
```

Query Q7 returns the titles of documents reached in one step from the Orsay Museum web server, the label of which matches "impressionistic". Query Q8 returns a list of ITEM elements, each one referencing (`URL` and `TITLE`) a document dealing with "painting" and retrieved by an index server (`about` operator).

3.3 Implementation

An SgmlQL interpreter (`http://www.univ-tln.fr/~gect/simm/SgmlQL`) implements all the functionalities described in this section. Initially, this tool was named MtSgmlQL because it has been developed within the MULTEXT european project, objective of which was to develop tools to anotate, analyze and manipulate corpora of SGML documents. SgmlQL have been developed for UNIX operating systems. It needs no preliminary indexing of queried documents. Queries are evaluated in a pipe-line mode, which provides a physical-level optimisation. Besides, it is possible to use SgmlQL as a filter in a UNIX command.

4 Integrating XGQL into SgmlQL

4.1 Introduction of environments in XGQL

In order to fully exploit the possibilities of the host language, the result-generation mechanism of XGQL (the "underscore" mechanism) must be enhanced to support environment generation. The underscore mechanism can be viewed as generating environments with a single variable. To support multiple-variable environments, the selected subexpressions in the query are no longer prefixed by an underscore character, but rather suffixed by a variable name in parentheses. In the concrete syntax, variable names are of the form `$name`. Then, we modify the definition of the result of a parse tree; instead of being a mere list of segments, $\rho(\Pi)$ is redefined as a mapping from variable names to lists of segments, as follows: Vertices in Π are considered "marked" by a variable x iff their associated expression is suffixed by (x). For each variable x occurring as a mark in Π, $\rho_x(\Pi)$ is defined as the singleton function mapping variable name x to the ordered (possibly empty) list of (possibly empty) segments $\sigma(v_1)$, $\sigma(v_2)$, ..., $\sigma(v_n)$, where v_1, v_2, ..., v_n is the ordered list of x-marked vertices encountered during a preorder traversal of Π. Finally, $\rho(\Pi)$ is defined as the union of the $\rho_x(\Pi)$ singleton functions, over all x occurring as a mark in Π. As before, $Q(\Delta)$ is defined as:

$$\cup_{\Pi \text{ is a } Q\text{-parse-tree for } \kappa(\Delta)} \{ \rho(\Pi) \}.$$

Members of $Q(\Delta)$ are distinct environments that are passed successively to the SgmlQL host language for processing. Observe that, again, $Q(\Delta) \neq \emptyset$ iff Q matches Δ, and that, if Q matches Δ but has no variables, $Q(\Delta) = \{\emptyset\}$, where $\emptyset$ can be seen as the "null environment".

4.2 Examples

We illustrate the proposed integration through examples. The first two use the document (`$doc`) containing bibliographical references (see section 3.2). They show user manipulations on structured documents. The three following queries are dedicated to natural language processing. The two last examples illustrate user search on Web pages. First, we rewrite queries Q4 (name and first surname of each author who published a paper in a conference held in 1999) and Q5 (papers

having more than five pages and published in a conference held in France). These new expressions are more compact and a pattern to filter is described in a single step.

(Q4')
```
select [$t, $a]
from xgql(<BIB> % <ENTRY> AUT*($a) TIT($t) % <DATE>[1999]</> % </> %
          </>) within $doc
```

As explained earlier in Section 2, an XGQL pattern can be viewed as a template that must match the document, but can match it in different ways, each generating a distinct environment. A "%" matches any well-formed (and possibly empty) segment. A variable name in parentheses ($id) after a subpattern will bind the variable to an element matching the subpattern. Any genID without <> represents an element with no content criterion (thus, is an abbreviation for <GENID>%</>). An expression like [1999] specifies a textual content criterion. In our example, each <ENTRY> element in the document will be matched separately, resulting in the generation of as many environments as the total number of <ENTRY> occurrences. Each environment will provide a binding for each of the variables $a, $t. The overall result of the query is a list of pairs ($t, $a) where $t is a TIT and $a is a list of AUT.

(Q5')
```
select element BIBENTRY
       attr: DATE = text($d)
       content: [element AUTHOR
                 content: [text($s), " ", text($n)], $t]
from xgql(<BIB> %
             <ENTRY> %
                <AUT> NAME($n) SURNAME($s) % </> % TIT($t)
                <CONF>[France]</> DATE($d) % FPAGE($p1) LPAGE($p2)
             </> %
          </>) within $doc
where (text($p2) - text($p1) + 1) > 5
```

In our example, each ENTRY element in the document will be matched separately, and each AUT element within each ENTRY element will also be matched separately, resulting in the generation of as many environments as the total number of AUT elements. Each environment will provide a binding for each of the variables $n, $p, $t, $d, $p1, $p2. Notice how the data extracted from the document is reorganized: the date becomes an attribute, the name and surname of each author are concatenated and become the content of a new element AUTHOR.

Consider now natural language processing for which text-oriented software tools are in dire need. Many NLP utilizations of corpora do pattern-matching on texts : for example, for modifying a markup or for extracting parts of text with a specific structure.

Suppose a book ($mybook) composed of paragraphs with sentences, where both paragraphs and sentences are marked up . The query Q9 extracts all the sentences of this book and removes all tags in these sentences. The query transforms the associated tree of the element $mybook.

(Q9)
```
replace xgql(: SENTENCE($s)) within $mybook
by element SENTENCE content : text($s)
```

Query Q10 modifies the marked up by describing the grammatical category as an attribute. For example by transforming <VERB>eat</VERB> into <WORD CAT = "VERB">eat</WORD>.

(Q10)
```
replace xgql(: SENTENCE($s))from $mybook
by replace xgql(<%> %($c) </>($w)) from $s
   by element WORD
      attr: CAT = gi($w)
      content:$c
```

The last query is taken from (Sornlertlamvanich, 1994), it makes it possible to search Thaï sentences which contains occurrences of the pattern N\CL--DET. This pattern represents sequences made of a noun consecutively or non consecutively followed by a classifier and consecutively followed by a determinant. Query Q11 extracts sentences with such a structure.

(Q11)
```
select $s
from xgql(: <SENTENCE>NOUN % CLASS DET </>($s))
within $mybook
```

Finally, we illustrate queries on HTML page. Suppose a document ($page) consists of a sequence of intermixed H1 and/or P elements. We want all paragraphs containing the word «painting», together with the nearest H1 element that precedes it. This query (Q12) will return all such paragraph, together with the nearest H1 element that precedes it.

(Q12)
```
select [$h, $p]
from   xgql( : H1 ($h) P* <P>[painting] </> ($p)) within $page
```

Query Q13 is the same but the paragraphs extracted must contain an image (format `.gif`).

(Q13)
```
select [$h, $p]
from xgql( : H1 ($h) P* <P>[painting] </> ($p)) within $page
where exists $t in xgql(:A) within $p: $t->HREF match "*.gif"
```

It is possible and can be useful to combine in a single query the XGQL pattern and SgmlQL pattern because the semantics of each one is clear and both return environments. For instance, the following query (Q14) adds further criteria to query Q7:

(Q14)
```
select [first TITLE within $d, $h, $p]
from  navigate node("http://www.navigo.com/paris/hist/musee-orsay.html")
              (-> as $l, _ as $d)
      xgql( : H1 ($h) P* <P>[painting] </> ($p)) within $d
where $l->label match "impressionistic"
      and exists $t in xgql(:A) within $p: $t->HREF match "*.gif"
```

returns the list of document titles reached from the Orsay museum Web server following one link, the label of which matches "impressionistic", and H1 element preceding P element which contains the word "painting" and at least one image.

5 Related works and conclusion

Our proposition is closed to the XML-QL language (Deutsch et al., 1998) which enables to extract, to transform and to integrate XML data. Its main operator where...construct is a particular form of the SgmlQL select...from...where. XML-QL patterns are element patterns which defines attributes and components to filter with a XML-like syntax, whatever their level (siblings, descendants or ancestors). It is not precised if a horizontal pattern with regular expressions - like TIT followed by AUT* - is allowed. Variables are used, as in SgmlQL, to extract fragments of documents which are post processed in the construct clause. XML-QL provides operators as group by or order by but not update operators as remove or replace.

Lorel (Abiteboul et al, 1997), the query langage of the data management system for semi-structured data Lore, is, as SgmlQL, derived on OQL. Patterns are expressed by the generalized path expressions with wild card and arbitrary regular expressions. Variables can be bound to patterns to extract parts of data. As in SgmlQL, most of patterns describing siblings can be expressed but by decomposition step by step and not in a single pass. Lorel does not provide an operator to build new documents from extracted parts and no update operators.

Community of natural language processing is interested in the querying of linguistic annoted corpora. Q4M (Mengel et al., 1998) of MATE project or LT NSL (1997) provide either interactive graphical interface to query XML documents or a C-based API for accessing and manipulating SGML documents. Even if they provide facilities, no one offers a declarative language, as SgmlQL+XGQL proposes, combining in the same and single expression, patterns to filter, conditions to verify and structure to build as result. We show in section 4 that if a document is finely marked up, it is possible to finely query or transform it. A typical query given for Q4M is : "Find all adverbs by Peter which include "H*" and follow directly after an answer by Mary. It can be expressed with "XGQL + SgmlQL" in an easy way.

Finally, we can cite the XSL(2000a) language, the XML(1998) Stylesheet Language. A stylesheet is a well-formed XML document defined to present a type of document and its purpose is not at all to provide a manipulation language. But XSL, by means of the language XSLT(1999), enables to transform the structure of a XML document before applying rules for its presentation. A rule is a pair of a pattern and its associated transformation. A pattern is described by means of an Xpath (1999) expression which captures any hierarchical or horizontal relationships between elements.

Any component of a XML document can be reached whatever its granularity. The Xpath expression is, according to us, a very interesting alternative to express patterns in the SgmlQL from clause. We study this integration in a parallel work. Conversely, XSLT is another very natural candidate host language for integrating the powerful patttern-matching of XGQL, an avenue we plan to explore in future work.

6 Bibliographical references

Abiteboul, S, Quass, D, McHugh J., Widow, A., Wiener J. (1997). The Lorel query language for semi-structured data. In *International Jouranl on Digital Libraries, 1(1)* (pp. 68—88).

Bruno ,E. , Le Maitre, J., Murisasco, E. (1999). "Controlled Hypertextual Navigation in the SgmlQL Language", Proceedings of 10th International Conference DEXA'99. Springer Florence, Italy.

Ceri, S.(1999). "Models and tools for designing data-intensive Web applications", *Actes des 15èmes journées Bases De Données Avancées*, Invited conference, Bordeaux.

Deutsch, A, Fernandez, M, Florescu, D., Levy, A., Suciu D. (1998). XML-QL: a query language for XML. Submission to the World Wide Web Consortium. Online: http://www.w3.org/TR/NOTE-xml-ql.

Dublin Core (2000). Dublin Core Metadata Initiative, online: <http://purl.org/dc>, visited 2000-02-28.

GILS (Global Information Locator Service) (1999), online: <http://www.gils.net/>, visited 2000-02-28.

Gonnet, G.H., Tompa, F.W. (1987). "Mind your grammar: a new approach to modelling text", *Proc. 13th VLDB Conference*, Brighton.

Harié, S., Le Maitre, J., Murisasco, E., Véronis, J. (1996). "SgmlQL, un langage de requêtes pour la manipulation de documents SGML", *Cahiers GUTemberg numéro spécial TEI (Text Encoding Initiative*), n°24 juin 1996, (pp 181-184).

Hopcroft, J.E., Ullman, J.D. (1979). *Introduction to automata theory, languages, and computation*, Addison-Wesley.

http://www.w3.org/TR/1999/NOTE-SYMM-modules-19990223

Kilpeläinen, P., Mannila, H. (1992). "Grammatical tree matching", *Proc. 3rd annual symp. Combinatorial Pattern Matching*, Springer-Verlag, pp. 162-174.

Kilpeläinen, P., Mannila, H. (1993). "Retrieval from Hierarchical Texts by Partial Patterns", *Proceedings of ACM-SIGIR'93*, Pittsburgh (US), pp. 214-222.

Kuikka, E., Salminen, A. (1997). "Two-dimensional filters for structured text." *Information Processing & Management*, vol. 33, no 1, 1997, pp. 37-54.

LT NSL (1997). Online: http://www.ltg.ed.ac.uk/software/lt_nsl.html

Marcoux, Y., Sévigny, M. (1997). "Querying hierarchical text and acyclic hypertext with generalized context-free grammars." In: L. Devroye, C. Chrisment. *Computer-Assisted Information Searching on Internet: Actes de la conférence RIAO 1997*, vol. 1. Paris, Centre de hautes études internationales d'informatique documentaire, 1997, pp. 546-561.

Mengel, A. Heid, U. Fitschen, A, Evert, S. (1998). Improved Query Language (Q4M). Online: http://www.ims.uni-stuttgart.de/projekte/mate/

Navarro, G., Baeza-Yates, R. (1995). "A language for queries on structure and contents of textual databases", *Proc. ACM-SIGIR'95*, Seattle.

Salminen, A., Tompa, F.W. (1992). "PAT expressions: an algebra for text search", *Proc. 2nd Int. Conf. on Computational Lexicography, COMPLEX'92*, pp. 309-332.

Sévigny, M. Marcoux, Y. (1996). "Construction et évaluation d'un prototype d'interface-utilisateurs pour l'interrogation de bases de documents structurés." *Canadian Journal of Information and Library Science / Revue canadienne des sciences de l'information et de bibliothéconomie*, vol. 21, no 3/4, septembre-décembre 1996, pp. 59-77.

SgmlQL reference manual and download. Online: http://www.univ-tln.fr/~gect/simm/SgmlQL/.

Sornlertlamvanich, V., Pantachat, W., Meknavin, S. (1994). Classifier Assignement by Corpus-Based Approach. In *Proceedings of the 15th Conference on Computational Linguistics* (Coling'94), Kyoto, Japan (pp. 556—561).

W3C (1998a). XML Extensible Markup Language (1998) 1.0, W3C Recommendation. Online: http://www.w3.org/TR/1998/REC-xml-19980210

W3C (1998b). Synchronized Multimedia Integration Language (SMIL) 1.0 Specification, W3C Recommandation 15-June-1998, online: <http://www.w3.org/TR/REC-smil>, visited 2000-02-28.

W3C (1999). Synchronized Multimedia Modules based upon SMIL 1.0. Online:

W3C (1999a). Xpath XML Path Language (1999), W3C Working Recommendation. Online: http://www.w3.org/TR/1999/REC-xpath-19991116

W3C (1999b). XSLT XSL tranformations (1999), W3C Working Recommendation. Online: http://www.w3.org/TR/1999/REC-xslt-19991116

W3C (2000a). XSL Extensible Stylesheet Language (2000), W3C draft. Online: http://www.w3.org/TR/2000/WD-xsl-20000112

W3C (2000b). XML Schema Part 1: Structures, W3C Working Draft 25 February 2000, online: <http://www.w3.org/TR/xmlschema-1/>, visited 2000-02-28.