AG-Based interactive system to retrieve information from XML documents

Alda Lopes Gançarski
University of Minho, Braga, Portugal, email: aldalopes@di.uminho.pt
Member of Laboratoire d’Informatique de Paris 6 (LIP6), Paris, France

Anne Doucet
Laboratoire d’Informatique de Paris 6 (LIP6), Paris, France,
email: Anne.Doucet@lip6.fr

Pedro Rangel Henriques
University of Minho, Braga, Portugal, email: prh@di.uminho.pt

Abstract

In this paper we describe a system to interactively access XML documents aiming at Information Retrieval. The system has two main modules: the query editor/processor, where the user specifies his needs; the document analyser, which performs operations for query evaluation. The interactive construction of queries is based on the manipulation of intermediate results during query edition and evaluation. Queries are written in IXDIRQL, a query language that extends XPath with selection operations to extract the interesting subset of elements from intermediate results. This helps the user in building queries to retrieve the desired results. Moreover, textual similarity search of traditional Information Retrieval is also possible in IXDIRQL, leading to a ranked list of elements.
To support a syntax-directed edition of queries and its incremental evaluation during the iterative process, IXDIRQL is specified by an Attribute Grammar (AG). This formalisation enables the use of an automatic generator of the desired working environment.

In our system, also documents are represented by AG. This representation uniformly defines structure, content and operations over documents; this allows for a better interoperability between components.

The system has been used by real users to check its correct behaviour and the correct specification of queries using selection operations.

 Contents

1 Introduction 3

2 IXDIRQL 8

3 The Attribute Grammar for a Structured Document 11
   3.1 Syntax of a Document 12
   3.2 Semantics of a Document 14
   3.3 Internal Representation of a Document 15

4 IXDIRQL Editor/Processor 16
   4.1 Syntax definition of IXDIRQL 18
   4.2 Semantics definition of IXDIRQL 19
   4.3 Editor characteristics 22
1 Introduction

When documents have the same kind of information, it is natural to think about describing the information in the same way for all of them. For example, business letters have almost all the same structure, so this structure should be described in a standard way. These ideas were at the origins of the Standard Generalized Markup Language (SGML) [10], a standard for specifying markup languages as a format of exchanging documents. The specification of the structural elements and their hierarchical relations for a given type of documents is made through the Document Type Definition (DTD). Later, the Hyper-text Markup Language (HTML) [24] was developed as an SGML application to show documents in the Web. This standard mainly makes use of presentation marks to describe how the textual parts are displayed by the browser. More recently, the eXtensible Markup Language (XML) [7] appeared as a simplifica-
tion of SGML adapted to exchange documents in the Web. XML documents have a description richer than the simple but limited one provided by HTML. An alternative way to DTD for specifying the structure of a collection of XML documents collection is the XML Schema language [11], which allows for a richer definition of data types.

The task addressed in this paper is the information retrieval from XML documents. Information Retrieval (IR) [4] consists of retrieving from a collection the relevant documents to a query, while returning as few as possible non-relevant documents. Moreover, the resulting documents should be ranked by their relevance to the query (i.e., by decreasing probability of being relevant). Usually, a query is a natural language expression describing the desired subject.

To take advantage from the structural information of XML documents, query formats for structured documents retrieval were enriched to access certain parts of documents. So, the user can access those parts based on content restrictions (comparing two element contents or an element content with a value) and structural restrictions (referring to elements and their structural relations).

The W3C Consortium is currently developing XQuery [6] to become the standard XML query language. XQuery is based on different query languages, such as XPath [5] and XML-QL [9]. All these query languages can be referred to as data retrieval (instead of information retrieval) languages because the retrieval system searches for all documents that satisfy the query [4]. Thus, there is no notion of relevance attached to the retrieval task. Some extensions to these languages appeared to allow for textual similarity restrictions [12, 8],
returning a ranked list of elements by their relevance. The W3C is proposing the Full-Text language [3] to extend XPath and XQuery with the possibility of associating a score (or relevance measure) to an expression that verifies if some phrase exists in some XML document component. Since 2002, the INEX initiative for the evaluation of XML retrieval takes place as an annual workshop to promote the evaluation of XML retrieval by providing a large test collection of XML documents, uniform scoring procedures, and a forum for organizations to compare their results [13].

However, query construction is not always an easy process because, among other reasons, the user may not have a deep knowledge of the query language, or may not know a priori exactly what to search; moreover he may realize that it is not exactly what was expected just when looking at the final result. To address this problem, IXDIRQL (Interactive XML Data and Information Retrieval Query Language) [16, 17] was defined as an extension of XPath, not only with textual similarity operations as in [12, 8], but also with the selection of the interesting parts of intermediate results and an interactive/incremental paradigm for building queries. With this paradigm, each operation specified by the user leads to intermediate results which the user can access. This helps him choosing the next operation, or changing an operation already introduced, or selecting interesting subsets in the intermediate results, until reaching the adequate query and thus the desired result.

Structured documents can be represented in a standard way using Attribute
Grammars (AG) [14]. Basically, an AG is a grammar extended with semantic
definitions by the use of attributes. DTD and XML Schema are more expressive
than grammars because they allow the definition of attributes to specify seman-
tic characteristics of XML elements. These attributes and respective values
are given in XML documents. AG allow the definition of other attributes not
appearing in the document, together with the corresponding evaluation rules.
So, it is desirable to represent structured documents using AG for the following
reasons:

- AG allow to represent the structure of any type of structured document.
- AG are expressive enough to represent different operations on documents,
  which enables for a better communication between different applications.
  It is important to notice that the same formalism (AG) is used both for
document structure and semantics representation.
- it is possible to automatically build a document analyser based on the
  corresponding AG using a generator like Lex/Yacc (for certain classes of
  AG) or LRC [19, 27].

Different works already used this representation in areas like structured doc-
ument databases [1, 2, 21, 22] and editing [25]. In [23], the authors propose this
representation to associate structured documents with a desired application-
dependant semantic definition. In our work, structured documents are repre-
sented by AG in a novel application domain: XML retrieval using IXDIRQL.
In this representation, semantics includes the evaluation process of structured
queries (*dynamic semantics*) and the verification of the correct reference to elements or XML attributes in path expressions (*static semantics*).

The XML representation model *Document Object Model* (DOM) [20] is proposed by W3C as a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. Differently, the AG representation does not allow to change the document but to make a variety of desired semantic operations.

This article presents the IXDIRQL system we developed and its use by real users. Figure 1 gives a schematic overview of the system’s architecture. The system has two main components: the IXDIRQL editor/processor, where the user interactively specifies queries (module 1), and the document AG analyser, which performs IXDIRQL operations over documents (module 2). Queries are introduced in the IXDIRQL editor and results are shown in a Mozilla view (module 4). The editor was built using LRC (module 3) based in the AG of IXDIRQL. Some operations to query evaluation require a direct access to documents. These operations are sent to the document analyser to perform them which, in turn, sends the results back to the editor. The document analyser was built using Lex/Yacc (module 6) based in the AG of the document collection. This AG results from a transformation of the DTD or the XML Schema of the collection using module 5.

In what follows, we start introducing IXDIRQL in Section 2 and the representation of structured documents by AG in Section 3. AG is an adequate
formalism to define the syntax and the semantics of a formal language. So, the formal definition of IXDIRQL is made through an AG. Representing documents and queries in the same way using AG allows the use of the same language processing techniques and tools. Thus, in a similar way to structured documents analyser, a structured editor and an incremental processor for queries can be built directly from this AG. Section 4 presents the IXDIRQL incremental editor we built from the AG of IXDIRQL. The document AG analyser is described in Section 5. Section 6 proposes a method to verify the correct understanding and use of selection operations and reports preliminary tests made. We finish the article with a conclusion and some directions for future work.

2 IXDIRQL

IXDIRQL is based on XPath to specify structural and content comparison restrictions. Basically, XPath allows the specification of elements by a path. For example, /article/title refers to titles inside articles. Filters impose conditions to elements, like article/author='Kevin'/title specifies titles of articles with an author called 'Kevin'. Several elements can be retrieved using groups, as in /article(title|author) to get the title and the author(s) of articles.

IXDIRQL extends XPath with the Sim textual similarity operator over a specified type of elements. Sim is independent from the relevance estimation method. For example, /article/title Sim 'XML' searches elements of type title being about 'XML'. The result is a list of titles ranked by their relevance.
Relevance measures are in [0..1]: 0 for non-relevant elements, 1 for relevant elements and values from 0 to 1 for different levels of textual similarity between the element and the natural language expression. Operations imported from XPath yield to a relevance of 1 because the elements completely satisfy the query restrictions. Logical operations imported from XPath are modified to return truth values, instead of Boolean values. Truth values are represented by the associated relevance $R$, meaning ‘true with probability $R$’. Thus, logical true is the truth value with relevance 1 and logical false is the truth value with relevance 0. Truth values with relevancies between 0 and 1 can happen in expressions like $\text{title Sim } 'XML' \text{ and } \text{author}= 'Kevin'$. Here, the result is a list of truth values which relevance values are calculated taking into account the relevance of titles with respect to the subject ‘XML’.

IXDIRQL also extends XPath with selection operations that consist of restricting intermediate results to the subset of elements that satisfy the user. There are three selection operators:

**Select** selects the subset of interesting elements based on some criteria that the user does not know how (or does not want) to specify by a filter. For example, in $\text{article/title Select } \{ tit_4, tit_8 \}$, the operator Select selects titles $\{ tit_4, tit_8 \}$ from the ones corresponding to the path $\text{article/title}$.

**SelectN n** selects a subset composed of the first $n$ elements\(^1\). For example, \( /\text{article/author}='Kevin' \text{ SelectN 10/title} \)

\(^1\)There is also the possibility of selecting the subset of the first $n$ elements using the predefined XPath function $\text{position()}$. For example, $/\text{article/author}='Kevin' \text{ SelectN 10/title}$
/article[author='Kevin'] SelectN 10 yields the first 10 articles of author 'Kevin' found in the collection. This can be interesting to have a final result with the appropriate size for further processing.

**JudgeRel** selects the subset of elements judged relevant by the user after using the operator Sim. For example, query article[title Sim 'XML']/reference gives a list of references ranked by the relevance of the title of the article where they are cited. To accept those references as relevant, the user must trust the relevance estimations made by the system. Otherwise, for each reference, he must access the title of the article where the reference is cited, judge it and, if relevant, accept the reference (if not, reject it). Using the **JudgeRel** operator, the user can judge and select relevant elements during query construction. Consequently, the final result is relevant and can be directly used for further processing. The previous query would then be article[title Sim 'XML'] JudgeRel \{tit_4, tit_8\}/reference, if tit_4 and tit_8 are judged relevant.

is equivalent to /article[author="Kevin"]/position()<10/title. However, this alternative requires from the user a deeper knowledge of XPath, including its pre-defined functions. **SelectN** is simpler and easier to understand which helps not expert users to specify their queries.
3 The Attribute Grammar for a Structured Document

An AG consists of a Context Free Grammar (CFG) extended with a set of attributes (and semantic rules for their calculation) which specify the semantics of the analysed text. If necessary, it also allows to impose contextual conditions to productions, based on attribute values. Contextual conditions correspond to the static semantics, in opposition to dynamic semantics. When an attribute value is calculated by the production where the respective symbol is derived (i.e. it is on the left-hand side of the production) it is called synthesized attribute. Otherwise, the attribute is called inherited. Synthesized attributes propagate the semantic information from the leaves up to the root of the syntax tree, while inherited ones do the same but from the root to the leaves or between sibling nodes.

In general, an AG is based on an abstract CFG where special words or characters of the language (like char in C programming language) are suppressed. The structural characteristics of the language and the textual components with a variable value are only taken into account. The abstract CFG is simpler and thus easier to process than the concrete (complete) one, that is why the AG is associated with it. From the concrete CFG, the AG processor parses a text. Then, knowing how to map the concrete CFG into the abstract CFG, the AG processor creates an Abstract Syntax Tree (AST) that represents the text. Nodes in an AST correspond to derived symbols. Leaves correspond to textual
blocks. Each node is identified by the symbol and the production from where it was derived. An AST is then decorated with attributes and their values according to the semantic rules of the AG. The result is a decorated AST (DAST) which represents the text and its meaning.

Using an AG, each document is represented by a DAST where internal nodes are elements, leaves are textual (or other media) content and attributes include the necessary transformations over the content and the structure such that the final documents meaning is the result of query operations evaluation.

Dynamic semantics of document AG includes, for each element, its unique identifier, the information about its XML attributes, the textual representation of its content and the result of an IXDIRQL operation that directly access documents, like searching for a certain type of element.

Next sections describe the syntax and the semantics of document AG by means of an example. For simplicity, only XML elements are treated and the semantics concerns only the IXDIRQL operation that searches for a certain type of element.

3.1 Syntax of a Document

When a document collection has an associated DTD (or Schema), the corresponding CFG is taken from the DTD using a set of mapping rules described in [15]. Elements and XML attribute definitions are transformed into symbols and productions of the CFG. This is done by the module 5 of Figure 1. To illustrate this, assume the following DTD Letter.
This DTD describes letters. Element declarations yield the following abstract CFG:

\[ p1 : \text{Letter} \rightarrow \text{Date Message} \]
\[ p2 : \text{Date} \rightarrow \text{TEXT} \]
\[ p3 : \text{Message} \rightarrow \text{TEXT} \]

Production \( p1 \) derives a letter. The most external element of a letter is \( \text{Letter} \) and it is derived by a non-terminal symbol with the same name. As specified in the DTD, \( \text{Letter} \) is derived in the sequence of symbols \( \text{Date} \), which derives a \( \text{Date} \) element, and \( \text{Message} \), which derives a \( \text{Message} \) element. Both \( \text{Date} \) and \( \text{Message} \) are textual elements. Thus, the corresponding productions \( p2 \) and \( p3 \), respectively, derive the terminal symbol \( \text{TEXT} \) which defines any string. Note that, being the abstract CFG, the opening and closing tags of elements are not derived.
3.2 Semantics of a Document

To define semantics, by convention, attribute names start by \( a \). An attribute \( a_{Att} \) of symbol \( S \) is written \( S.a_{Att} \). Synthesized (resp. inherited) attributes are preceded by \( \uparrow \) (resp. \( \downarrow \)).

The IXDIRQL operation that searches for elements of a certain type in a document returns a set of elements of the desired type. This set is stored in the attribute \( a_{Result} \) associated to non-terminal symbols that derive elements. Each element is identified by the natural number stored in attribute \( aId \). For the simple example DTD \( \text{Letter} \) of section 3.1, identifiers are easy to evaluate. Suppose that the identifier given to the last element of the last analysed document is \( \text{previousId} \). Then, element \( \text{Letter} \) of the next document to be analysed has attribute \( aId \) with value obtained by incrementing \( \text{previousId} \) by 1. The next elements appearing in the same document correspond to symbols \( \text{Date} \) and \( \text{Message} \). For these elements, \( \text{previousId} \) is incremented by 2 and 3, respectively.

Assuming that \( \text{desiredType} \) represents the type of elements to search, semantic definitions of the document \( AG \) are the following:

\[
\begin{align*}
p1 : \text{Letter} & \rightarrow \text{Date Message} \\
& \quad \uparrow \text{Letter.aId} = \text{previousId} + 1 \\
& \quad \uparrow \text{Letter.aResult} = \text{If} (\text{desiredType} = \text{'Letter'}) \\
& \quad \quad \text{Then} \\{ \uparrow \text{Letter.aId} \} \cup \uparrow \text{Date.aResult} \cup \uparrow \text{Message.aResult} \\
& \quad \quad \text{Else} \\uparrow \text{Date.aResult} \cup \uparrow \text{Message.aResult} \\
p2 : \text{Date} & \rightarrow \text{TEXT}
\end{align*}
\]
The attribute \textit{aResult} is synthesized and is calculated by adding the identifier of the present element (attribute \textit{aId}) to the resulting set (attribute \textit{aResult}) if its type is the desired one.

### 3.3 Internal Representation of a Document

Consider the following example document of type \textit{Letter} associated with the example DTD of section 3.1:

```xml
<Letter>
  <Date: 2005/10/20 15:44:38 >14 February 2005</Date>
  <Message> Hi, Marianne! It is vacations. Bye!</Message>
</Letter>
```

Using the AG defined in section 3.2, this document is represented by the DAST of Figure 2. Synthesized attributes are shown on the right-hand side of
the corresponding non-terminal symbol and inherited attributes on the left-hand side. Dependencies between attributes with respect to semantic rules are shown by dashed line arrows. Each symbol and respective attributes are surrounded by a dashed rectangle. As we can see, attribute $aResult$ is synthesized and thus calculated in a bottom-up fashion.

4 IXDIRQL Editor/Processor

The IXDIRQL editing environment must allow the user accessing intermediate results of the query operations. Besides, it should: (1) Help the user to write his queries by making explicit the syntax of the language and also its static semantics, such as which elements are valid operands for each operation. Elements validation is based on the documents DTD, as depicted in Figure 1. (2) Be associated with an incremental processing of the operations. This means that, each time a new operation is inserted or an existing one is changed, the system does not calculate all the query operations. Instead, it first calculates the intermediate results of the new or changed operation; then, it recalculates the intermediate results that are dependent on the previous ones and the final result of the query. (3) Allow for full-text querying as an alternative to interactive construction for expert users.

Instead of building an editor with these characteristics for IXDIRQL from scratch, it was automatically created associated with an incremental processor (module 1 in Figure 1). For that, we used LRC [19], a generator of incremental
environments based on formal definition of languages through AG (module 3 in Figure 1). Thus, the AG of IXDIRQL was defined for two reasons: (1) A formal definition of IXDIRQL is needed as it is the basis to build a correct language processor implementation. To formally define the syntax and the semantics of a formal language, such as IXDIRQL, an AG is an adequate formalism. (2) The AG of IXDIRQL can be directly used by LRC to generate an incremental editor/processor with the desired three mentioned above characteristics. The generated language processor analyses texts as described in section 3 for general AG analysers. LRC accepts AG written in the *Synthesizer Specification Language* (SSL). SSL also allows for the specification of different ways to visualize the DAST of a text. For example, the same structural representation through a DAST of an XML document can be shown in different views with its own tags, with tags converted to HTML, or even without tags and an adequate indentation of textual content. Interesting editing possibilities can also be defined in SSL, like menus associated with alternative productions of a non-terminal symbol or buttons associated with semantic actions. This can improve the editing/processing quality and speed.

The following sections explain how the AG of IXDIRQL is formed, showing some example productions. For a better comprehension, instead of SSL syntax, we will use BNF notation which is widely known and easily adapted to SSL. The conventions of Section 3 are still valid.
4.1 Syntax definition of IXDIRQL

In this section we present the abstract CFG and the concrete CFG of IXDIRQL.
For simplification, we only present productions related to the non-terminal symbol \textit{Operation}. This symbol derives structural operations, namely set operations (union, intersection, negation), logic operations (disjunction, conjunction) and paths. A path accesses a type of elements in XML documents. For example, the path \textit{article/section} accesses sections inside articles.

In the concrete CFG, \textit{Operation} is defined by the following productions that correspond to the different structural operations, respectively union, intersection, negation, disjunction, conjunction and path:

\begin{align*}
  pc1 : & \text{Operation} \rightarrow \text{Operation} \ ackslash \ 	ext{Operation} \\
  pc2 : & \mid \text{Operation} \ 'int' \ 	ext{Operation} \\
  pc3 : & \mid \ 'not' \ 	ext{Operation} \\
  pc4 : & \mid \text{Operation} \ 'or' \ 	ext{Operation} \\
  pc5 : & \mid \text{Operation} \ 'and' \ 	ext{Operation} \\
  pc6 : & \mid \text{Path}
\end{align*}

The corresponding productions in the abstract CFG are obtained by removing the special strings that denote the IXDIRQL operation, like $\text{int}$ for an intersection. Thus, we have:

\begin{align*}
  pa1 : & \text{Operation} \rightarrow \text{Operation} \ \text{Operation}
\end{align*}
These productions are complemented with a set of precedence and associativity rules that are taken into account by the generated parser.

4.2 Semantics definition of IXDIRQL

The dynamic semantics of IXDIRQL corresponds to the evaluation of a query. At each point of the query there is a context corresponding to the set of elements from which the system will reach new desired ones. For example, let 'introduction | conclusion' be a query which returns introductions and conclusions of articles. Suppose the documents collection is composed of articles $a_1$ and $a_2$, with introductions $i_1$ and $i_2$, respectively, and conclusions $c_1$ and $c_2$, respectively. To calculate the result of the query, the context is the set of articles $\{a_1, a_2\}$. Taking these identifiers, the system looks for introductions and conclusions. It then associates article $a_1$ to the result $\{i_1, c_1\}$ and article $a_2$ to the result $\{i_2, c_2\}$.

To illustrate semantic rules of the AG for query processing, consider the derivation of a union by the abstract CFG (production $pa1$ in Section 4.1). Attributes $aContext$ and $aRes$ of symbol $Operation$ correspond to the context and
the result of an operation, respectively. Their semantic rules are:

\[\text{pa1 : Operation } \rightarrow \text{ Operation Operation} \]

\[\downarrow\text{Operation}_2.a\text{Context} = \downarrow\text{Operation}_1.a\text{Context} \]

\[\downarrow\text{Operation}_3.a\text{Context} = \downarrow\text{Operation}_1.a\text{Context} \]

\[\uparrow\text{Operation}_1.a\text{Res} = \text{FUnion}(\uparrow\text{Operation}_2.a\text{Res}, \uparrow\text{Operation}_3.a\text{Res}) \]

Each occurrence of symbol \textit{Operation} is referred to associating an integer suffix to indicate its position in the production with respect to the others. The context of an operation is passed through the inherited attribute \textit{aContext} from \textit{Operation}_1 to both \textit{Operation}_2 and \textit{Operation}_3, in order to calculate the corresponding results. These results are stored in the synthesized attribute \textit{aRes} and, then, used as arguments of function \textit{FUnion} to calculate the result of \textit{Operation}_1, which consists of a union.

Assume the example query ‘\textit{introduction} | \textit{conclusion}’ shown above, together with the same collection of articles \textit{a}_1 and \textit{a}_2. This query is represented by the partial DAST of Figure 3. The derivation trees of both child occurrences of \textit{Operation} are omitted for simplification. They correspond to path operations (‘\textit{introduction}’ and ‘\textit{conclusion}’, respectively) derived by production \textit{pa6}. A context is represented by a rectangle which encloses its elements. Results are represented in a table that associates each element of the context with the respective set of elements found. Each element is a tuple formed by its unique identifier (\textit{a}_1, \textit{a}_2, ...) and its relevance at the each moment of the query (rel-
evance is 1 if there are no textual similarity operations). In this DAST, it is clear that synthesized attributes \( aRes \) are calculated in a bottom-up fashion and inherited attributes \( a\text{Context} \) in a top-down fashion.

The **static semantics** of IXDIRQL checks the correct construction of paths, by analysing the type of elements which are specified and their respective hierarchical relations. A path operator ('/' for child and '//' for descendant) is derived from symbol \( Path\text{Operator} \) and a type of element to search is derived from symbol \( Element \). The productions that derive an entire absolute path\(^2\), i.e. a list of a path operator followed by a type of element to search, are:

\[
\begin{align*}
pa7 & : Path \rightarrow Path\text{Operator} \ Element \ Path \\
pa8 & : \ | \ \epsilon
\end{align*}
\]

Assume the following attributes: attribute \( aPath\text{Operator} \) of symbol \( Path\text{Operator} \) stores the derived operator ('/' or '//' ); attribute \( a\text{Element} \) of symbol \( Element \) stores the type of element to search; attribute \( a\text{ContextTypes} \) of symbol \( Path \) stores the type of elements of the context. Calculation rules are omitted for simplification. When deriving \( Path \) on the left-hand side of production \( pa7 \), the context is the set of elements from which the system will search the next desired type of element, derived by \( Element \) on the right-hand side.

Static semantics is, then, defined by the contextual condition (CC) defined \(^2\)Absolute paths start with a path operator. Relative paths have no path operator at the beginning. Depending on the position of a relative path in the query, the system adequately assumes a child or a descendant operator.
for production pa7:

\[
CC : \text{If } (\langle \text{PathOperator.aPathOperator} = '/' \rangle) \\
\quad \text{Then } \text{IsChild}(\langle \text{Element.aElement} \rangle, \langle \text{Path}1.aContextTypes \rangle) \\
\quad \text{Else } \text{IsDescendant}(\langle \text{Element.aElement} \rangle, \langle \text{Path}1.aContextTypes \rangle)
\]

According to the path operator, predicates IsChild and IsDescendant check if the type of the desired element is child or descendant, respectively, of some of the types of element in the context. If the condition is not verified, an error is shown to the user in the editor.

4.3 Editor characteristics

The editor generated by LRC based on the IXDIRQL AG (module 1 in Figure 1) is structured: the edition is guided by the syntax of the language. Non-terminal and pseudo-terminal symbols are associated with places for introducing information. Such a place consists in an expression, between \langle and \rangle, that suggests the type of information to introduce. When a non-terminal symbol has alternative productions, the user chooses the one he wants using a menu as in Figure 4. In this figure, a structured operation (\langle Operation? \rangle) can be one among six shown in the menu. If the user chooses an intersection, the editor becomes the one in Figure 5, where one of the operands is already introduced (article/ref).

All terminal symbols are automatically shown in the right place, like the
operator $\$int\$ in Figure 5.

**IXDIRQL static semantics** consists of checking the consistency of element types in paths with respect to the hierarchical relations between them. For example, in Figure 5, there is a semantic error in path `/article/ref` because `ref` elements are not child of `article` elements. This type of errors are placed after the correspondent operation to help the user correcting the query immediately.

The characteristics of the editor guide the user with respect to the syntax and static semantics of the language. However, it is convenient for the user to have a certain knowledge of IXDIRQL to write his queries, like knowing what is a path.

### 4.4 Presentation of query results

In the system, results are presented in a browser for structured documents (we choose *Mozilla*, module 4 in Figure 1). Results are ordered by decreasing order of relevance of elements or truth values. If relevancies are equal, the order in which elements appear in the collection is used. Figure 6 is an example of a result. In the results view, each element is presented with its unique identifier (which can be used for subsequent selection operations), its relevance, a pointer to the respective document and its content. Truth values are shown with their relevance, the identifier of the corresponding element (for example, the element associated with the filter where a Boolean operation is made) and a pointer to the document corresponding to this element.

The access to intermediate results is made through a *control view* shown
in Figure 7. The user can access the results by answering the questions that appear in this view. If the user wants to see the results, they are displayed in the results view.

4.5 Relevance estimations

In our system, relevance measures of textual similarity operations are based on the traditional IR vector space model. We used this model because it is simple and fast, being superior or almost as good as a large variety of alternative ranking methods [4].

Traditional information retrieval

In traditional IR, documents and queries are represented by the set of terms which are considered to describe the meaning of the corresponding text. The selection of the terms is called automatic indexing and consists of extracting them from words encountered in documents after some operations that reduce the size of the index and improve results quality. First, a lexical analysis is done to distinguish the words from not interesting characters, like punctuation marks. Then, some words, referred to as stopwords, are not included in the index because they do not treat any specific subject. Examples are articles, prepositions and conjunctions. Elimination of stopwords reduce index size about 40%. Finally, words are reduced to their stem, which is the portion of a word which is left after the removal of its affixes (i.e., prefixes and suffixes).

A term with a high frequency in a document is considered adequate to repre-
sent that document. Let \( \{d_1, ..., d_N\} \) be a collection of documents; \( \{t_1, ..., t_T\} \) a set of terms; \( n_i \) the number of documents where term \( t_i \) appears \( (i=1, ..., T) \); \( freq_{ij} \) the frequency of the term \( t_i \) in document \( d_j \) \( (j=1, ..., N) \). The normalized frequency \( tf_{ij} \) of term \( t_i \) in document \( d_j \) is given by:

\[
    tf_{ij} = \frac{freq_{ij}}{\max_{k=1, ..., T} freq_{kj}}
\]

Terms which appear in a small fraction of the collection are good discriminants for that fraction. The discriminate power of a term \( t_i \) is given by:

\[
    idf_i = \log N/n_i
\]

The multiplication of these two measures \((tf*idf)\), referred to as \( tf.idf \), is frequently used to represent the terms of the collection for relevance estimations.

Documents and natural language expressions of queries can be represented by a vector having as components the correspondent \( tf.idf \) measures of terms. In this vector space model [26], the relevance of a document with respect to a query is the result of a correlation function between the corresponding vectors. A simple, efficient, and thus frequently used correlation function is the cosine which gives the relevance of document \( d_j \) \( (j=1..N) \) with respect to query \( q \) by:

\[
    sim(d_j, q) = \frac{d_j \times q}{|d_j| \times |q|}
\]

Let \( w_{ij} \) be the measure \( tf.idf \) of term \( t_i \) \( (i=1..T) \) in vector \( d_j \) \( (j=1..N) \). The relevance is, then:

\[
    sim(d_j, q) = \frac{\sum_{i=1}^{T} w_{ij} \times w_{iq}}{\sqrt{\sum_{i=1}^{T} w_{ij}^2} \times \sqrt{\sum_{i=1}^{T} w_{iq}^2}}
\]

25
Structured documents retrieval

To adapt relevance computation to IXDIRQL queries, tf.idf measures are taken from the set of elements of the collection, instead of whole documents. Let $Ne$ be the number of elements in the collection; $n_i$ the number of elements where term $t_i$ appears ($i=1..T$); $freq_{ij}$ the frequency of term $t_i$ in element $e_j$ ($j=1..Ne$). The tf.idf representation is now called $tf.ief$ where:

$$tf_{ij} = \frac{freq_{ij}}{\max_{l=1..T} freq_{lj}} \quad \text{and} \quad ief_i = \log Ne/n_i.$$  

To combine relevancies from different query operations, relevancies are interpreted as probabilistic events, as done in different works including some participating in INEX, like [18]. The following probabilistic formulas for disjunction and conjunction of events are, then, used:

$$P(p_1 \lor \ldots \lor p_n) = \sum_{i=1}^{n} (-1)^{i-1} (\sum_{1 \leq j_1 < \ldots < j_i \leq n} P(p_{j_1} \land \ldots \land p_{j_i}))$$

$$P(p_1 \land \ldots \land p_n) = P(p_1) \times \ldots \times P(p_n) \text{ assuming that } p_1 \ldots p_n \text{ are independent}$$

An event is the fact that an element is relevant. The relevance is the probability associated with the event, i.e. the probability that an element is relevant. For example, in the query `/article Sim 'XML'/reference Sim 'XSL'` there are two similarity operations, one for articles, the other one for references. Suppose that article $a_1$ has relevance $P(a_1)$ interpreted as the probability that $a_1$ is relevant with respect to 'XML'. Suppose also that reference $r_1$ is cited in article $a_1$
and has relevance \( P(r_1) \). \( P(r_1) \) is the probability that \( r_1 \) is relevant with respect to the subject 'XSL'. In the result, both relevancies must be combined to get the final relevance for each reference. This is done by \( P(a_1 \land r_1) = P(a_1) \times P(r_1) \) which is the probability that article \( a_1 \) and reference \( r_1 \) are both relevant.

When \( Sim \) is followed by \( JudgeRel \), the relevance associated to the operator \( Sim \) is first calculated. Then, the relevance of each element selected by \( JudgeRel \) becomes 1, and the relevance of the remaining elements becomes 0.

5 The AG-based Analyser for Structured Documents

Document AG analyser (module 2 in Figure 1) represents each document by a DAST. DASTs are constructed in two steps. The first one corresponds to the evaluation of attributes that do not depend on the query, such as attribute \( aId \) in the AG of section 3.2. These attributes are evaluated before query session for faster query response. We call the result of the first step the partial DAST.

From the DTD (or schema) of the collection, the document AG generator (module 5 in Figure 1) automatically generates the document AG for the first step of DAST construction. This is done using rules that map the schema DTD into the documents CFG extended with application specific semantics definitions (IXDIRQL operations in our case). The document AG is, then, used as input to a generator of language analysers. As the present AG is accepted by Lex/Yacc (module 6 of Figure 1), we used it to generate the first step document.
AG analyser. Yacc calculates attribute values during the parsing of documents. Consequently, the IXDIRQL AG was defined from the concrete CFG.

The second step of DAST construction uses attributes of the partial DAST to complete IXDIRQL operations over documents during query evaluation. All these operations need the construction of a set of elements based on some conditions, such as being of a certain type. In the example of section 3.2, the attribute \textit{aResult} is calculated in the second step using attribute \textit{aId} from the partial DAST. For fast query processing, instead of creating an attribute evaluator for any class of AG, we built a simple left-to-right, bottom-up visit to the partial DAST, parameterized with the conditions for an element to become part of the result.

To optimize IXDIRQL operations that search for a certain type of element, we defined \textit{a priori} different visits to the partial DAST, each one for a different type of element. This avoids visiting nodes that can not be part of the result.

### 5.1 Storing documents representation

Partial DASTs of documents are stored in a \textit{structure index}. This index is a file where information concerning nodes is stored making a left-to-right, bottom-up visit to each DAST. In this way, when the document AG analyser completes the partial DAST with the attributes to calculate query results, it visits the partial DAST by reading the file sequentially, starting from the beginning. The structure index has the information shown in Figure 8 (a). Each element (\textit{elemId}) is associated with its father (\textit{fatherId}), the semantic information of its
AG attributes (like attribute \textit{aId}) and a pointer to the next element of the same type to perform visits to the partial DAST by type of element (\textit{Semantics}).

The semantic information concerning the textual representation is stored separately. In traditional IR, information about vector model textual representation of a collection of documents (terms and their tf.idf values, as explained in section 4.5) is stored in an index for a faster query response. This index is called \textit{inverted file}. An inverted file consists of a \textit{vocabulary} file and of a \textit{postings} file. The vocabulary stores information about terms. As shown in figure 8 (b), each term (\textit{term}) is associated with its ief (\textit{ief}) and a pointer to its information in the postings file (\textit{ptrPost}). As shown in Figure 8 (c), the postings file stores, for each term, the element identifiers where the term appears (\textit{elemId}) and the respective tf measure (tf).

Element relevancies from \textit{Sim} operations are estimated using both the structure and the textual indexes.

6 Testing the system with real users

It is interesting to verify if users understand the meaning and the need of selection operations to perform certain types of information needs. This can be tested observing real users working with the prototype. Notice that an evaluation of system’s performance is not in the scope of this paper.

To make the tests about the use of selection operations, some information needs are defined so that the result implies the use of such operations in queries,
unless the results are changed manually. Let us assume that we use the INEX collection [13]. It is composed of 12107 articles of the IEEE Computer Society during the period 1995-2002. The associated DTD defines, among others, the types of elements and the corresponding hierarchical relations of Figure 9. These elements occur, for example, in the document of Figure 10. An example of an information need is *To search for references cited in articles with a title, which you consider being related to ‘XML’*. The corresponding query is:

```
article[fm//atl Sim 'XML' JudgeRel {...}]//bb
```

In this query, titles (of articles) about ‘XML’ are returned using the operator *Sim*. Then, *JudgeRel* reduces these titles to those judged relevant by the user. So, references found are indeed cited in articles with a title that the user considers to be related to ‘XML’.

Facing the pre-defined information requests, queries specified by the users belong to one of the following 4 cases:

- **A**: the query includes a selection operation and satisfies the information request;
- **B**: the query does not include a selection operation and does not satisfy the information request;
- **C**: the query includes a selection operation and does not satisfy the information request;
- **D**: the query does not include a selection operation but satisfies the information request because the user manually modified the results.
The frequency of queries in each of these cases give an indication of the good understanding and use of selection operations.

**Preliminary tests**

Preliminary tests were made with five users and three pre-defined information requests (given in appendix). Users were selected among the researchers in computer science of the laboratory where this work took place, including four PhD students and one professor, all knowing well XML, four of them knowing well XPath and one knowing XPath vaguely. IXDIRQL was presented in a document with explanations concerning similarity and selection operations. The document collection was a subset of the INEX collection composed of 100 articles from the *Intelligent Systems* journal, years 2000 and 2001. Articles have between 110 and 2568 elements. Both Figures 9 and 10 were included in the document given to the users, together with explanations. The system was also presented to each user, executing some example queries.

Users behaviour includes aspects that we cannot control, like the concentration in executing the experience, the knowledge level about XPath, the understanding of the document describing the experience (which includes explanations about IXDIRQL, the collection of documents and the information requests). Despite this, we could observe the following facts:

- There were no doubts about the document explaining IXDIRQL and the
information requests. In general, users got well familiarised with the interface.

- To read the document and learn the interface, users took about 20 minutes; reading the information needs and specifying the queries took about 30 minutes.

- There are no queries of type C. All the queries where a selection operation was made satisfy the information request, i.e. 100% of selection inclusions are correct. This shows that the users always knew why selection was used.

- The query type D never occurred because no user changed manually a result. It is expected that a system to search information returns the desired result, without subsequent modification.

- Finally, the most interesting fact is that, among the 15 queries made, 11 are of type A, i.e. in 73% of queries users correctly include selection operations (detailed information about the use of each selection operation is given in appendix).

Despite the small number of users involved in this experiment, the results presented here give an indication of users behaviour and tendency when using selection operations.

7 Conclusion and future work

In this paper, we present a system to make interactive information retrieval from XML documents using IXDIRQL. This language extends XPath with textual similarity operations and selection operations to select the interesting subset
of elements from intermediate results during query construction. This helps users to specify the query that yields the desired result. Besides, the access to intermediate results guides the user in choosing the next operation or changing one already introduced. This interactive and iterative way of query construction helps in building the queries when users want to retrieve information in a large collection of documents without having a complete knowledge of what they are looking for. It also helps users that do not have a deep knowledge of the query language. For a better performance, the evaluation of query operations is incremental so that, after some change in the query, only the computations depending on that change are made.

The test of the system by real users showed not only its correctness, but also gave an indication that there is a tendency to correctly use selection operations when they are necessary to fulfil an information request.

In the system, documents are represented by AG. This representation is richer than the one provided by DTDs or Schemas since it allows to integrate, not only structure and content, but also operations over documents in the same representation. Moreover, a document analyser was automatically created giving the document AG to Lex/Yacc.

As future work, we intend to perform the following sequence of tasks:

1. The generated editor has characteristics that we consider suitable to perform the preliminary tests reported in Section 6. However, system’s per-
formance can be improved including a button in the editor to access to intermediate results and adding the “query by example” mechanism to help users building queries, avoiding the need for a deep knowledge of the query language.

2. In order to verify the interest of selection operations and the access to intermediate results, using the method suggested in Section 6, to perform large scale tests concerning number of involved users, document collection size and number of pre-defined information requests.

3. To extend XQuery with textual similarity operations and selection operations, as we did from XPath to IXDIRQL. An adequate processing system can be built using the IXDIRQL editor/processor adapted with the extended XQuery functionalities.

4. To design a methodology to use the extended XQuery in order to take into account ontologies associated with documents to improve the retrieval process.

Acknowledgements

The authors are grateful to the Portuguese Fundação para a Ciência e a Tecnologia for the financial support. We also would like to thank the technical support given by João Saraiva.
References


Appendix

Information requests

1. Search for references cited in 2 articles of an author whose first name is “Giovanni”.

2. Search for references cited in articles with a title, which you consider being related to ‘XML’.

3. Taking the references obtained in 2., search for the authors of those ones with an interesting title for you (at least one title).

Queries satisfying these information requests are, respectively:

1. \texttt{article[fm//fnm='Giovanni'] SelectN 2//bb}
   \texttt{article[fm//fnm='Giovanni'] Select \{id_1, id_2\}//bb}

2. \texttt{article[fm//atl Sim 'XML' JudgeRel \{\ldots\}]//bb}

3. \texttt{article[fm//atl Sim 'XML' JudgeRel \{\ldots\}]//bb[atl Select \{\ldots\}]//au}
   \texttt{article[fm//atl Sim 'XML' JudgeRel \{\ldots\}]//bb[atl Sim '...' JudgeRel \{\ldots\}]//au}

In the first query, \texttt{SelectN} reduces the number of articles of author “Giovanni” to 2. Another possibility is to make the selection using \texttt{Select} applied to 2 article identifiers \{id_1, id_2\}.

The second query is given in Section 6 together with the corresponding information request.

Finally, the third query includes the operator \texttt{Select} to select the titles of references that are interesting to the user. Alternatively, the user can search for
titles about some interesting subject using \textit{Sim} and select with \textit{JudgeRel} those ones he considers relevant.

\textbf{Results}

Table 1 gives the relation between the three information requests and the five users using the classification from A to D of the queries and, eventually, the selection operation used.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
\textbf{Request/User} & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & \textit{SelectN} & \textit{Select} & \textit{SelectN} & \textit{SelectN} & \textit{Select} \\
2 & \textit{JudgeRel} & \textit{JudgeRel} & B & B & B \\
3 & \textit{JudgeRel} & \textit{Select} & B & \textit{Select} & \textit{Select} \\
\hline
\end{tabular}
\caption{The classification of queries made by the users.}
\end{table}

In the 15 queries made by the users, there are five possibilities of using \textit{SelectN} (in information request 1), ten possibilities of using \textit{Select} (in information requests 1 and 3) and ten possibilities of using \textit{JudgeRel} (in information requests 2 and 3). The frequencies verified are 2/5 for \textit{SelectN} (40\%), 6/10 for \textit{Select} (60\%) and 3/10 (30\%) for \textit{JudgeRel}. \textit{Select} is the most used operator, which may indicate that users understand it better or find it more interesting or easier to use than the other ones.
Figure 1: The proposed system.

Figure 2: The DAST of the document example of type *Letter*. 
Figure 3: The partial DAST of query 'introduction | conclusion'.

Figure 4: IXDIRQL editor showing a menu.

Figure 5: IXDIRQL editor showing an intersection.

Figure 6: Results view.

Figure 7: Control view.

Figure 8: Structure index (a), vocabulary (b) and postings (c).
Figure 9: Some element types and hierarchical relations from INEX DTD.

<article>...
  <fm>...
    <atl>What's Next for the E-Book?</atl>
    <au><fnm>Giovanni</fnm><snm>Flammia</snm>...</au>...
  </fm>...
  <bb id="bibx10181">
    <au><fnm>B.N.</fnm><snm>Schilit</snm>...</au>
    <atl>"As We May Read: The Reading Appliance..."</atl>
  </bb>
  <bb id="bibx10182">
    <au><fnm>B-W.</fnm><snm>Chang</snm>...</au>...
    <au><fnm>J.</fnm><snm>Mackinlay</snm>...</au>...
    <atl>"Fluid Links for Informed and Incremental..."</atl>...
  </bb>...
</article>

Figure 10: A sample document