Indexing and Querying Linguistic Metadata and Document Content

Niraj Aswani and **Valentin Tablan** and **Kalina Bontcheva** and **Hamish Cunningham**[*]
Department of Computer Science
University of Sheffield
Regent Court, 211 Portobello Street
Sheffield, S1 4DP, UK
{niraj,valyt,kalina,hamish}@dcs.shef.ac.uk

Abstract

The need for efficient corpus indexing and querying arises frequently both in machine learning-based and human-engineered natural language processing systems. This paper presents the ANNIC system, which can index documents not only by content, but also by their linguististic annotations and features. It also enables users to formulate versatile queries mixing keywords and linguistic information. The result consists of the matching texts in the corpus, displayed within the context of linguistic annotations (not just text, as is customary for KWIC systems). The data is displayed in a graphical user interface, which facilitates its exploration and the discovery of new patterns, which can in turn be tested by launching new ANNIC queries.

1 Introduction

The need for efficient corpus indexing and querying arises frequently both in machine learning-based and human-engineered natural language processing systems. A number of query systems have been proposed already and (Christ 94), (Mason 98), (Bird *et al.* 00a) and (Gaizauskas *et al.* 03) are amongst the most recent ones. In this paper, we present a full-featured annotation indexing and retrieval search engine, called ANNIC (ANNotations-In-Context), which has been developed as part of GATE (General Architecture for Text Engineering) (Cunningham *et al.* 02).

Whilst systems such as (McKelvie & Mikheev 98), (Gaizauskas *et al.* 03) and (Cassidy 02) are targeted towards specific types of documents, (Christ 94), (Bird *et al.* 00a) and (Mason 98) are general purpose systems. ANNIC falls in between these two types, because it can index documents in any format supported by the GATE system (i.e., XML, HTML, RTF, e-mail, text, etc). These existing systems were taken as a starting point, but ANNIC goes beyond their capabilities in a number of important ways. New features address issues such as extensive indexing of linguistic information associated with document content, independent of document format. It also allows indexing and extraction of information from overlapping annotations and features. Its advanced graphical user interface provides a graphical view of annotation mark-ups over the text along with an ability to build new queries interactively. In addition, ANNIC can be used as a first step in rule development for NLP systems as it enables the discovery and testing of patterns in corpora.

Section 2 introduces the GATE text processing platform which is the basis of this work. Following this, we briefly describe how Lucene is used to index documents (Section 3). This section also provides details of the ANNIC implementation and the changes made in Lucene.

2 GATE

GATE is a large-scale infrastructure for natural language processing applications (Cunningham *et al.* 02). Lingustic data associated with language resources such as documents and corpora is encoded in the form of annotations. GATE supports a variety of formats including XML, RTF, HTML, SGML, email and plain text. In all cases, when a document is created/opened in GATE, the format is analysed and converted into a single unified model of annotation. The annotation format is a modified form of the TIPSTER format (Grishman 97) which has been made largely compatible with the Atlas format (Bird *et al.* 00b), and uses 'stand-off markup' (Thompson & McKelvie 97). The annotations associated with each document are a structure central to GATE, because they encode the language data read and produced by each processing module. Each annotation has a start and an end offset and a set of features associated with it. Each feature has a name and a relative value, which holds the descriptive or analytical information such as Part-of-speech and sense tags, syntactic analysis, named entities identifica-

[*]This work was partially supported by an AHRB grant ETCSL and an EU grant SEKT.

tion and co-reference information etc.

JAPE, Java Annotation Patterns Engine, is part of the GATE system. It is an engine based on regular expression pattern/action rules over annotations. JAPE is a version of CPSL (Common Pattern Specification Language). This engine executes the JAPE grammar phases - each phase consists of a set of pattern/action rules. The left-hand-side (LHS) of the rule represents an annotation pattern and the right-hand-side (RHS) describes the action to be taken when pattern found in the document. JAPE executes these rules in a sequential manner and applies the RHS action to generate new annotations over the matched regular expression pattern. Rule prioritisation (if activated) prevents multiple assignments of annotations to the same text string.

This paper demonstrates how ANNIC indexes GATE processed documents with their annotations and features and enables users to formulate versatile queries using JAPE patterns. The result consists of the matching texts in the corpus, displayed within the context of linguistic annotations (not just text, as is customary for KWIC systems). The data is displayed in a graphical user interface, which facilitates its exploration and the discovery of new patterns, which can in turn be tested by launching new ANNIC queries.

3 Apache Lucene

ANNIC is built on top of the Apache Lucene[1] a high performance full-featured search engine implemented in Java, which supports indexing and search of large document collections. Our choice of IR engine is due to the customisability of Lucene.

Lucene document is a basic unit of indexing and search operations. All information associated with Lucene documents is stored in units called fields, where each field has its name and a textual value. (e.g. contents, url, modified date etc.). *Analyzer* knows what to parse and how to convert text into the format that *Index Writer* understands. Index Writer builds a *Token Stream (a sequence of words)*, which describes information about the token text. *Token* contains linguistic properties, and other information such as the start and end offsets and the type of the string (i.e. the lexical or syntactic class that the token belongs to). We will use the term *Lucene token*

[1]http://lucene.apache.org

Table 1: Lucene Token Generation

Lucene Token	Position Increment
John	1
wants	1

Table 2: Lucene Token Generation

Lucene Token	Position Increment
John	1
wants	1
want	0

to refer to the tokens created by Lucene. *Filters* take the *stream of tokens* as input and add or delete Lucene tokens in the token stream. For example, a stemmer would add a new Lucene token with base word for each word that is not in its base form and a stop word filter would remove all stop words from the token stream so that they do not get indexed. Not only Lucene provides the ability to create user defined queries through its API, it also supports a wide range of predefined queries. This includes wild character queries, boolean queries, phrase queries etc.

Every Lucene token has its own *position* in the *token stream.* This position remains relative to its previous Lucene token and is stored as a *position increment factor* in the token stream. Consider the example in Table 1 and Table 2 which show the strings, the Lucene tokens derived from them, and their respective position increments in the token stream. Executing a stemmer over the above sentence would generate two extra words, which are stored with 0 increments immediately after the word they refer to in the token stream.

Along with its position increment attribute, each Lucene token in the token stream comprises of four attributes: text (e.g. wants), start offset, end offset and type (e.g. word). Lucene stores only the first attribute (i.e. text) in its indices.

When a Lucene query is submitted to the Lucene query parser, an array that contains hits is returned as a result. Each hit is an object that contains a pointer to the document, in which one or more patterns have been found, and the score of that hit. Documents in this hit array are organized in a descending order of their scores, i.e. the most relevant document appears first. This arrangement allows users only to refer to the n number of top most documents in the results.

4 ANNIC

The aim of the ANNIC system is to index the linguistic information and other metadata and retrieve the annotation patterns in the form of KWIC concordances (see 5). After few changes in the behaviour of the key components of Lucene, we were able to make Lucene adaptable to our requirements.

4.1 Lucene Token generation

As mentioned before, Lucene only indexes the text attribute of a Lucene token. To meet our requirements, i.e. to index the linguistic information and metadata, Lucene was modified to index also the type attribute. Type attribute holds a string assigned by lexical analyzer that defines the lexical or syntactic class of the Lucene token. GATE documents need to be separated into tokens by a tokeniser (*GATE Token* from now on) before they get indexed with ANNIC. This is required as tokens are the basic segments of any document and therefore they should be indexed in order to perform full-text search. Every annotation in GATE has a corresponding features associated with it. We create a separate Lucene token for every feature in the document. In the case where multiple annotations and their features refer to the same text in the document, we use the "Position increment" attribute to indicate their positions. Consider the following example:

E.g. the word Bill is annotated as:

```
    GATE Token
     POS: NNP
     Kind: word
     String: Bill
 Person
```

Table 3 explains the token stream that contains tokens for the above annotations. The annotation type itself is stored as a separate Lucene token with its attribute type * and text as the value of annotation type. This allows users to search for a particular annotation type. In order not to confuse features of one annotation with others, feature names are qualified with their respective annotation type names. Where there exist multiple annotations over the same piece of text, only the position of the very first feature of the very first annotation is set to 1 and it is set to 0 for the rest of the annotations and their features. This enables users to query over overlapping annotations and features.

It is possible for two annotations to share the same offsets. They can share either start, end or both offsets. The built-in GATE annotation comparator is used for this purpose. First, the start offsets are compared and then the end offsets. If comparator returns two annotations as sharing both offsets, such annotations are kept on the same position in the token stream, and otherwise one after another. This may lead to a problem. What if annotations overlap each other (i.e. they share only one of the start and end offsets)? In this case, though annotations do not appear one after another, they are stored one after another. This may lead to incorrect results being returned and therefore the results are verified in order to filter out invalid overlapping patterns.

Before indexing GATE documents with Lucene, we convert them into the Lucene format and refer to them as GATE Lucene documents. In order to fetch patterns for their left and right contexts, it was necessary for some old concordances programs to have all documents available at the search time (Mason 98). This may lead to serious performance penalties. To overcome this problem, the token stream is stored in a separate file as a Java serializable object in the index directory. Later, it is retrieved in order to fetch left and right contexts of the found pattern.

4.2 Gate Query Parser

JAPE patterns support various query formats. Below we give few examples of JAPE patterns. Actual patterns can also be a combination of one or more of the following pattern clauses:

1. *String*
2. {*AnnotationType*}
3. {*AnnotationType == String*}
4. {*AnnotationType.feature == feature value*}
5. {*AnnotationType1, AnnotationType2.feature == featureValue*}
6. {*AnnotationType1.feature == featureValue, AnnotationType2.feature == featureValue*}

Order of the annotations specified in ANNIC query is very important. In Lucene, document must contain the specified keywords, no matter in which order they exist. Order is important only for the phrase queries. Since the default implementation of Lucene indexer indexes only the

Table 3: Token stream entries for the word Bill annotated as Token and Person

Sr. No.	Lucene Token Text	Lucene Token Type	Pos. Incr.	Description
1	Token	*	1	Annotation Type Token
2	NNP	Token.pos	0	pos feature with value NNP
3	word	Token.kind	0	kind feature with value word
4	Bill	Token.string	0	string feature with value Bill
5	Person	*	0	Annotation Type Person

text attribute of Lucene Token, it does not allow searching over the type attribute. Certain characters used in JAPE patterns have different meanings in Lucene. E.g. Lucene uses { } (opening and closing brackets) to recognize the range queries and these characters are used to enclose the annotation type in JAPE. Lucene query parser does not support position increments in queries. For example if one wants to search for annotations of type *Location* and *Person* referring to the same piece of text, Lucene does not support this. On the other hand, the respective JAPE pattern would be {*Location, Person*}.

JAPE patterns also support the | (OR) operator. For instance, {*A*} *(*{*B*} | {*C*}*)* is a pattern of two annotations where the first is an annotation of type *A* followed by the annotation of type either *B* or *C*.

Due to the various reasons explained above, we introduce our own query parser (ANNIC Query Parser) which accepts JAPE queries. Instead of comparing only the text attribute of Lucene Token, we also compare the type attribute. Lucene query parser, before accessing index, converts each keyword into an instance of *Term* class and compares them with the terms in index. Table 4 demonstrates how JAPE pattern tokens are converted into query terms. In order to use predefined Lucene queries (i.e. Boolean and Phrase queries), JAPE patterns with *OR* operator are normalized into the *AND normalized form* and all such patterns are *OR*ed together to form a Boolean query.

Lucene Phrase query considers its each token as a separate *term* and sets its position to the previous terms position + 1. This behaviour leads to a problem in the context of JAPE queries. For example, user issues the following query:

{*Location, Person.gender = male*}

This should search for the text that is annotated as *Location* and *Person*, where the *Person* annotation must contain a feature called *gender* with value *male*. In this case, the ANNIC query parser creates two separate terms (*Location* and *Person.gender = male*). In order to make both terms referring to the same location, positions of these terms must remain same. If the position of first term is *n*, Lucenes phrase query implementation makes the position of second term to *n+1*. This results into a pattern where the first annotation is *Location* and is followed by the annotation *Person.gender = male*. To overcome this problem, one solution is to pass customized term positions along with terms to the phrase query. Given a term and its position respective to its previous term, Lucene searches within its index to find the term only at the given position. Thus, instead of searching the second term at the *n+1* position, Lucene seeks a term that occurs at *n* position. This disables automatic increment in term's position and also allows searching for the overlapping annotation.

But even after this arrangment, there exists one major overlapping problem. For example for the text "Mr. Tim-Berners Lee told ...", where the text "Mr." is annotated as "Title", "Tim-Berners" as "FirstName", "Lee" as "Surname", "Mr. Tim-Berners Lee" as "Person" and finally "told" as "Token" with the part-of-speech tag "verb". For these annotations, the tokens "Title" and "Person" will be placed at the same position in the token stream, while "FirstName", "Surname" and "Verb" will be placed one after another after the "Title" and the "Person" annotations. This results into incorrect results when the query is : {Person} {Token.string == "told"}. When searching this pattern in the token stream, "Person" is not followed by the Token string "told", instead "Person" is followed by the annotation "FirstName", which is followed the annotation "Surname" and which is followed by the "told". To solve this problem, after converting the JAPE query into the Lucene query terms, we issue the query that contains only the initial terms which refer to the same location. For example, instead of querying with {Person}{Token.string == "told"}, we query index with {Person}. As a result this query returns all positions from the

Table 4: JAPE pattern tokens and their respective Query terms

	Query Term	
JAPE Pattern Token	**Term Text**	**Term Type**
String	String	Token.string
{annotationType}	annotationType	*
{annotationType.featureType == value}	value	annotationType.featureType

Table 5: Klene Characters

Query	**Interpretation**
({A})+3	({A}) | ({A}{A}) | ({A}{A}{A})
{B}({A})*3	({B}) | ({B}{A}) | ({B}{A}{A}) | ({B}{A}{A}{A})
{B}({A} | {C})+2	({B}{A}) | ({B}{C})| ({B}{A}{A}) | ({B}{A}{C}) | ({B}{C}{A}) | ({B}{C}{C})

token stream where the annotation is "Person". We compare the rest terms (i.e. "Token.string == "told") by fetching terms after the "Person" annotation and by comparing query terms with them.

Annotations in left and right contexts: As described earlier, each token stream referring to a separate document in the corpus is stored in a separate file as a Java serializable object and is retrieved once the Lucene tokens matching the query results in the token stream are known. Along with a list of documents, positions (i.e. where these annotations in the token stream appear) are also retrieved. This helps in skipping to a specific location in a token stream and reduces the lookup time. Numbers of tokens, specified in a context window field at run-time, are also fetched from the token stream before and after the pattern so as to show them as the left and right contexts in the GUI.

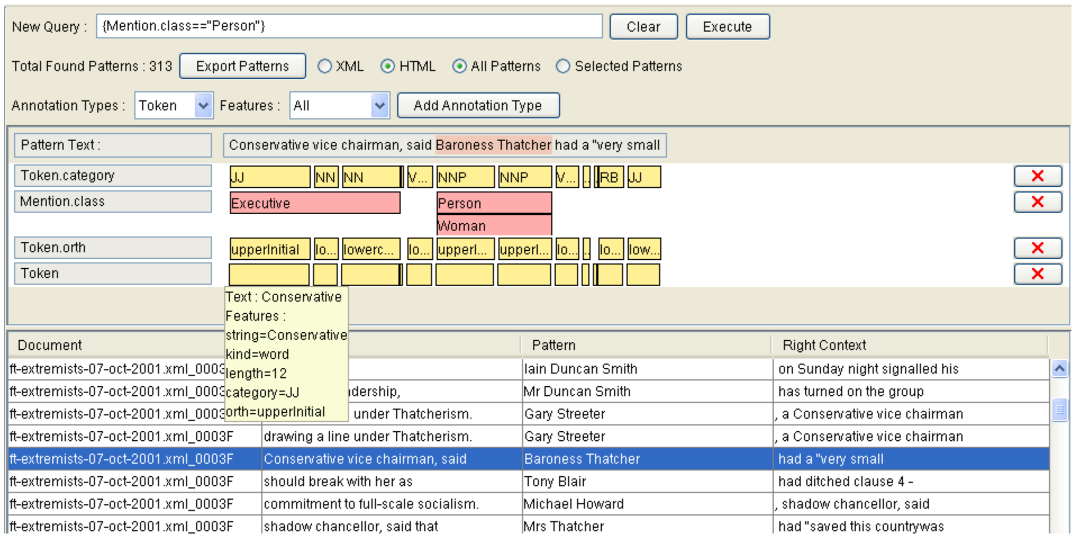
Klene operators: ANNIC supports two operators, + and *, to specify the number of times a particular annotation or a sub pattern should appear in the main query pattern. Here, *({A})+n means one and up to n occurrences of annotation {A} and ({A})*n* means *zero or up to n occurrences of annotation {A}*. Table 5 lists few example queries to illustrate the use of klene characters.

5 ANNIC user interface

ANNIC provides an advanced user interface at the presentation layer that allows users to index a large collection of documents (i.e. corpus), search indices and analyze the found patterns along with their left and right contexts concordances. At indexing time, the user can specify the corpus to be indexed, the annotation type that acts as document tokens, annotation set which contains the annotations to index, features and annotation types not to include in index and finally the location of index on the local or network file system. At search time, the user specifies the maximum number of documents to retrieve as results, number of tokens to show in the left and right contexts and finally the JAPE pattern query.

5.1 ANNIC Viewer

Figure 1 gives a snapshot of an ANNIC search window. The bottom section in the window contains the patterns along with their left and right context concordances and the section at top shows graphical visualization of annotations. ANNIC shows each pattern in a separate row and provides tool tip that shows the query that the selected pattern refers to. Along with its left and right context texts, it also lists the name of documents that the patterns come from. When the focus changes from one pattern to another, graphical visualization of annotations (GVA, above the pattern table) changes its current focus to the selected pattern. Here, users have an option of visualising annotations and their features for the selected pattern. The figure shows the highlighted spans of annotations for the selected pattern. Annotation types and features can also be selected from the drop-down combo box and their spans can also be highlighted into the GVA. When users choose to highlight the features of annotations (e.g. Token.category), GVA shows the highlighted spans containing values of those features. Whereas when users choose to highlight the annotation with feature all, ANNIC adds a blank span in GVA and shows all its features in a popup window when mouse enters the span region. A new query can also be generated and executed from the ANNIC GUI. When clicked on any of the highlighted spans of the annotations, the respective query clause is placed in the *New Query* text box. Clicking on *Execute* issues a new query and refreshes the GUI output. ANNIC also provides


New Query : {Mention.class=="Person"} Clear Execute

Total Found Patterns : 313 Export Patterns XML [✓ HTML] [✓ All Patterns] Selected Patterns

Annotation Types : Token Features : All Add Annotation Type

Pattern Text :	Conservative vice chairman, said Baroness Thatcher had a "very small
Token.category	JJ NN NN V... NNP NNP V... RB JJ
Mention.class	Executive Person Woman
Token.orth	upperInitial lo... lowerc... lo... upperI... upperI... lo... lo... low...
Token	

Text : Conservative
Features :
string=Conservative
kind=word
length=12
category=JJ
orth=upperInitial

Document		Pattern	Right Context
ft-extremists-07-oct-2001.xml_0003		Iain Duncan Smith	on Sunday night signalled his
ft-extremists-07-oct-2001.xml_0003	dership,	Mr Duncan Smith	has turned on the group
ft-extremists-07-oct-2001.xml_0003	under Thatcherism.	Gary Streeter	, a Conservative vice chairman
ft-extremists-07-oct-2001.xml_0003F	drawing a line under Thatcherism.	Gary Streeter	, a Conservative vice chairman
ft-extremists-07-oct-2001.xml_0003F	Conservative vice chairman, said	Baroness Thatcher	had a "very small
ft-extremists-07-oct-2001.xml_0003F	should break with her as	Tony Blair	had ditched clause 4 -
ft-extremists-07-oct-2001.xml_0003F	commitment to full-scale socialism.	Michael Howard	, shadow chancellor, said
ft-extremists-07-oct-2001.xml_0003F	shadow chancellor, said that	Mrs Thatcher	had "saved this countrywas

Figure 1: ANNIC Viewer

an option to export results in XML or HTML files with options of all patterns and selected patterns.

6 Applications of ANNIC

ANNIC is used as a tool aiding the development of JAPE rules. Language engineers use their intuition when writing JAPE rules trying to strike the ideal balance between specificity and coverage. This requires them to make a series of informed guesses which are then validated by testing the resulting ruleset over a corpus. ANNIC can replace the guesswork in this process with actual live analysys of the corpus. Each pattern intended as part of a JAPE rule can be easily tested directly on the corpus and have its specificity and coverage assesed almost instantaneously.

ANNIC can be used also for corpus analysys. It allows querying the information contained in a corpus in more flexible ways than simple full-text search. Consider a corpus containing news stories that has been processed with a standard named entity recognition system like ANNIE[2]. A query like {Organization} ({Token})*3 ({Token.string=='up'}|{Token.string=='down'}) ({Money} | {Percent}) would return mentions of share movements like "BT shared ended up 36p" or "Marconi was down 15%". Locating this type of useful text snippets would be very difficult and time consuming if the only tool available were text search. ANNIC can also be useful in helping scholars to analyse linguistic

Table 6: ANNIC queries

QP	Patterns
1	{Token.string==Microsoft} | "Microsoft Corp"
2	{Person} {Person}
3	{Person} {Token.category==IN} {Token.category==DT})*1 {Organization}
4	({Token.orth==allCaps} | {Token.orth==upperInitial}) ({Token.kind==number,Token.length==1})+2 {Token.kind==number,Token.length==1}) ({Token.orth==allCaps} | {Token.orth==upperInitial})
5	({Token.kind==number})+4 ({Token.string ="/"} | {Token.string=="-"}) ({Token.kind==number})+2 ({Token.string=="/"} | {Token.string=="-"}) ({Token.kind==number})+2
6	{Title} ({Token.orth==upperInitial} | {Token.orth==allCaps}) ({FirstPerson})*1
7	{Token.category=="DT"} ({Token.category=="NNP"} | {Token.category=="NNPS"}) ({Token.category=="NNP"} | {Token.category=="NNPS"})
8	({Token.category=="DT"})*1 {Location} {Token.category=="CC"} ({Token.category=="DT"})*1 {Location}
9	{Token.category=="IN"} ({Token.category=="DT"})*1 {Location} {Token.category=="CC"} ({Token.category=="DT"})*1 {Location}
10	{Organization}{Token.category=="IN"} ({Token.category=="DT"})*1 {Location}

QP=Query Pattern

[2]GATE is distributed with an IE system called ANNIE, A Nearly-New IE system.

Table 7: ANNIC query results

	BNC 10%		HSE		NEWS	
QP	**ST**	**P**	**ST**	**P**	**ST**	**P**
1	11.276	112	0.5	0	1.252	3
2	24.798	17	2.0	0	0.933	12
3	5.23	6	7.0	6	0.432	2
4	24.33	24	26.458	14	0.803	0
5	50.139	264	110.738	39	6.652	36
6	39.029	238	120.054	180	12.37	1038
7	99.813	480	192.013	321	16.854	1261
8	62.971	81	126.823	124	5.508	281
9	52.08	43	96.735	67	3.672	134
10	6.191	10	11.875	5	0.692	11

QP=Query Pattern,ST=Search Time,P=Patterns

corpora. Sumerologists, for instance, could use it to find all places in the ETCSL corpus [3] where a particular pair of lemmas occur in sequence.

7 Performance Results

In order to evaluate the performance of ANNIC, we experimented on three different corpora (large, medium, and small), processed with GATE: 10% of the BNC (British National Corpus)(374 documents,1443.84MB), HSE (Health and Security Experiments)(192 documents,896MB), and finally the NEWS corpus (446 documents, 39.4MB).

We tested the performance with several types of queries: string only queries, combinations of strings and linguistic data, and patterns with quantified Klene operators. Table 6 lists some of the different types of queries which were issued over the indexed corpuses. Table 7 gives the statistics of output of these queries. It provides different statistics including the time taken by ANNIC to retrieve the results and the number of patterns retrieved.

8 Related Work

(McKelvie & Mikheev 98) describe a suite of programs, LT INDEX, that supports indexing of large SGML documents. It indexes elements by their position in the document structure and by their textual content. ANNIC is more generic, because it can cope with a wider range of formats, while covering the same functionality.

CUE (Corpus Universal Examiner) system (Mason 98) splits the corpus data into different data streams (e.g. actual words, POS information), which are stored along with their positioning information in the index. Unlike CUE, ANNIC maintains a fixed structured data format (Term string, Term type, position) within indices and converts all annotations and their features into this consistent format. (Christ 94) describes separate layers for their corpus query system, where index access is described at the physical layer; interpreting user queries, searching within indices and processing of results at the logical layer; and the graphical user interface at the presentation layer. Their system is aimed at indexing all text documents that their modules at the physical layer can convert into a predefined format. Similarly ANNIC also indexes any document format that is supported by the GATE. Lucene and GATE both play a vital role in carrying out the tasks at physical layer. GATE reads different kinds of documents (SGM, EML, MAIL, XHTM, RTF, XML, SGML, HTML, TXT etc.) from a file system or from the web and transforms them into GATE documents, which are then processed by the ANNIC via the GATE API. ANNIC then converts them in a format that Lucene can index and store. GATE, ANNIC and Lucene work altogether at the logical layer. GATE processes the documents and provides an API that helps ANNIC to deal with document text, annotations and their features. ANNIC takes queries from users, interprets them using the query parser and submits them to Lucene. Once the results are out, ANNIC uses respective token streams stored under the index directory to fetch the patterns and left and right contexts along with their annotations to prepare the GUI.

(Gaizauskas *et al.* 03) describe a system, XARA that indexes any well-formed XML document. It combines an indexer, a server and a windows client. Indexer requires information like how element content is to be tokenized and how tokens are to be mapped to index terms etc. ANNIC supports not only XML but many other types of documents supported by the GATE. Similar to XARA, in ANNIC as well, the decision of how documents be tokenized is left on a user (e.g., GATE supplies tokenisers for several languages).

In order to investigate new models for semi structured data that are appropriate to XML, (Buneman *et al.* 98) describes a query language that is beyond any XML query languages. They describe extraction rules that consist of expressions along the tree and are expressed using the HTML Extraction Languages (HEL). Their query

[3]http://www-etcsl.orient.ox.ac.uk/

language comes with navigation operators, regular expressions and conditions to retrieve information even from the nested structures. ANNIC query parser works on top of the GATE annotations and features and supports search over overlapping annotations and features. Its advanced user interface allows users to visualize the nested structure of the annotations with their features highlighted.

(Kazai *et al.* 04) discuss the overlapping problem in content-oriented XML retrieval. They discuss the INitiative for the Evaluation of XML Retrieval (INEX) system, which discusses the matrices to evaluate the XML retrieval results. Their argument is that if in an XML document, a sub element satisfies a content-oriented query, parent element would also satisfies the same query. Thus, instead of including only a subcomponent in the result, INEX also includes the parent component. In ANNIC, the overlapping problem, as discussed in (Kazai *et al.* 04), does not exist due to two reasons. 1) Annotations in GATE documents are stored as an annotation graph. Thus comparing the structure of XML documents where elements contain texts, in GATE documents annotations are created over the text. 2) ANNIC queries are very specific about the annotation types, i.e. query itself describes the annotation type in which the string should be searched. If user does not specify annotation type, ANNIC does it automatically to search strings with the GATE token annotation type.

References

(Bird *et al.* 00a) S. Bird, P. Buneman, and W. Tan. Towards a query language for annotation graphs. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, Athens, 2000.

(Bird *et al.* 00b) S. Bird, D. Day, J. Garofolo, J. Henderson, C. Laprun, and M. Liberman. ATLAS: A flexible and extensible architecture for linguistic annotation. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, Athens, 2000.

(Buneman *et al.* 98) P. Buneman, A. Deutsch, W. Fan, H. Liefke, A. Sahuguet, and W.C. Tan. Beyond XML Query Languages. In *In Proceedings of the Query Language Workshop (QL'98)*, 1998.

(Cassidy 02) S. Cassidy. Xquery as an annotation query language: a use case analysis. In *Proceedings of 3rd Language Resources and Evaluation Conference (LREC'2002)*, Gran Canaria, Spain, 2002.

(Christ 94) O. Christ. A Modular and Flexible Architecture for an Integrated Corpus Query System. In *Proceedings of the 3rd Conference on Computational Lexicography and Text Research (COMPLEX '94)*, Budapest, 1994. `http://xxx.lanl.gov/-abs/cs.CL/9408005`.

(Cunningham *et al.* 02) H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02)*, 2002.

(Gaizauskas *et al.* 03) R. Gaizauskas, L. Burnard, P. Clough, and S. Piao. Using the XARA XML-Aware corpus query tool to investigate the METER Corpus. In *In Proceedings of the Corpus Linguistics 2003 Conference*, pages 227–236, Lancaster, UK, 2003.

(Grishman 97) R. Grishman. TIPSTER Architecture Design Document Version 2.3. Technical report, DARPA, 1997. `http://www.itl.nist.gov/div894/-894.02/related_projects/tipster/`.

(Kazai *et al.* 04) G. Kazai, M. Lalmas, and A. Vries. The Overlapping problem in Content-Oriented XML Retrieval Evaluation. In *Proceedings of the 27th International conference on Research and development in information retrieval*, pages 72–79, Sheffield, UK, 2004.

(Mason 98) O. Mason. The CUE Corpus Access Tool. In *Workshop on Distributing and Accessing Linguistic Resources*, pages 20–27, Granada, Spain, 1998. `http://www.dcs.shef.ac.uk/~hamish/dalr/`.

(McKelvie & Mikheev 98) D. McKelvie and A. Mikheev. Indexing SGML files using LT NSL. LT Index documentation, from `http://www.ltg.ed.ac.uk/`, 1998.

(Thompson & McKelvie 97) H. Thompson and D. McKelvie. Hyperlink semantics for standoff markup of read-only documents. In *Proceedings of SGML Europe'97*, Barcelona, 1997.