Master Thesis

An XQuery information extraction library

Author(s):
Schaufelberger, Michael

Publication Date:
2010

Permanent Link:
https://doi.org/10.3929/ethz-a-006117067

Rights / License:
In Copyright - Non-Commercial Use Permitted
An XQuery Information Extraction Library

Master’s Thesis
Michael Schaufelberger

Supervised by:
Prof. Dr. Donald Kossmann
Dr. Peter Fischer

Systems Group, http://systems.ethz.ch/
Department of Computer Science
ETH Zurich

January – July 2010
Abstract

While most current search engines and content management systems are based on keywords, semantic approaches to manage and search data are becoming more and more popular. A lot of effort has been made for the past few years towards “Information Extraction” rather than “Information Retrieval”.

The goal of this thesis is to make information extraction functionality available to XQuery, a functional programming language designed to process XML data. Since adding machine-readable semantics often involves RDF or other kinds of XML reformatting, the advantages of such functionality in connection with XQuery are obvious.

By introducing information extraction related work, this thesis investigates the advantages and disadvantages of such systems with regard to the requirements that arise from our goal. This information together with possible use cases then lead us to the definition of an API for the XQuery library. Finally, we discuss the structure, functionality and performance of our implementation and point out some topics for future work.
Acknowledgments

My gratitude goes to my supervisors Prof. Donald Kossmann and Dr. Peter Fischer for their help and feedback during the whole project. In particular, I would like to thank Prof. Donald Kossmann for his ideas that lead the thesis in the right direction and Dr. Peter Fischer for all his support and efforts especially with implementation and MXQuery issues. Furthermore I very much appreciated the participation of Maria Grineva and Kyumars Sheykh Esmaili in several meetings, sharing their thoughts and ideas. Last but not least, I would like to thank Stephan Merkli for proofreading my thesis.
# Contents

1 Introduction  
1.1 Background and Motivation  
1.2 Problem Statement  
1.3 Contributions  
1.4 Thesis Overview

2 Overview of Information Extraction Systems  
2.1 Requirements  
2.1.1 Usability  
2.1.2 Good Out of the Box Performance  
2.1.3 Coreference Resolution  
2.1.4 Support for Multiple Languages  
2.1.5 Extensibility  
2.1.6 Openness  
2.1.7 Community and Active Development  
2.2 Reference Text  
2.3 Reviews  
2.3.1 MarkLogic Server  
2.3.2 MinorThird  
2.3.3 OpenCalais  
2.3.4 GATE  
2.4 Summary

3 API Design  
3.1 Use Cases  
3.1.1 Extraction of Person Entities  
3.1.2 Entity Annotations  
3.1.3 Extraction of Relations  
3.1.4 Other Examples  
3.2 API Requirements  
3.3 API Definition  
3.3.1 Overview  
3.3.2 Data Model  
3.3.3 Functions  
3.3.4 GATE-Specific API

4 Implementation  
4.1 Overview
Contents

4.2 GATE Implementation ........................................... 21
  4.2.1 GATE’s Java API ........................................... 21
  4.2.2 EasyGate – An Abstraction of GATE’s Java API .......... 23
  4.2.3 MXQuery Extension ....................................... 25
  4.2.4 XQuery Module ........................................... 25
4.3 OpenCalais Implementation .................................... 26
  4.3.1 OpenCalais REST API .................................... 26
  4.3.2 XQuery Module ........................................... 26

5 Evaluation .......................................................... 29
  5.1 Functionality ................................................... 29
    5.1.1 Person Extraction ....................................... 29
    5.1.2 E-Mail Address Annotation .............................. 30
    5.1.3 Contact Extraction ...................................... 30
    5.1.4 Other ..................................................... 32
  5.2 Performance ................................................... 32
    5.2.1 Increasing the Document Size ......................... 32
    5.2.2 Increasing the Number of Documents ................. 34

6 Conclusion .......................................................... 39
  6.1 Summary ....................................................... 39
  6.2 Future Work ................................................... 39
    6.2.1 Facilitate Relationship Extraction .................... 39
    6.2.2 Revise the EasyGate Class ............................. 40
    6.2.3 Provide Higher-Level Functions ....................... 40
    6.2.4 Support for Multiple Documents ..................... 40
    6.2.5 Evaluation of the Information Extraction Quality ... 40
    6.2.6 Classification of Information Extraction Systems ... 40

A How to Use the Library ........................................... 41
  A.1 Download and Installation ................................. 41
  A.2 Usage ......................................................... 42

B Other Use Case Examples ......................................... 43
  B.1 Wikipedia Links ............................................. 43
  B.2 Twitter Topics ............................................... 44
Chapter 1

Introduction

1.1 Background and Motivation

An extensive amount of information created by humans is stored in unstructured or semi-structured ways. Data on the World Wide Web, in e-mails or company-internal documents often contain mostly plain text. Common approaches to manage and search in this kind of data are keyword-based content management systems or search engines respectively. However, more modern approaches show a tendency to do “Information Extraction” rather than classic “Information Retrieval”.

For the past few years a lot of effort has been made to develop systems that semantically analyze and annotate unstructured data. Although the interest for such systems in research has already been there in the nineties [CGW95] [CL96], information extraction is becoming more and more popular as recently as these days, considering projects like the Europe Media Monitor [BvB+05] or OpenCalais [Reu10a].

When extracting entities and relations from unstructured data, most systems use some kind of XML format for annotations or descriptions (e.g. RDF). Furthermore, additional information in semi-structured documents often is present as XML. Thus having information extraction functionality in a programming language that is designed to process XML, like XQuery, is clearly worthwhile.

1.2 Problem Statement

Due to the popularity of information extraction and the prevalent use of XML in that area, the main goal of this thesis is to provide key functionality for information extraction with XQuery. To get there we first investigate existing information extraction systems and specify possible use cases. In a second step, we define an API that allows to implement these use cases as easily as possible and at the same time is generic enough, so that possible implementations are interchangeable without affecting the definition of the API.
1.3 Contributions

Based on a list of requirements we compared different existing information extraction systems and chose those that best meet our needs. We defined an API whose functionality can be covered by the chosen systems and with which the previously specified use cases can be implemented. Along with the API we also defined a data model that is compatible with the chosen systems.

Regarding code contributions we implemented the defined API as an XQuery library in a way that its underlaying information extraction system is interchangeable. We implemented a port to GATE [CMBT02] by writing an MXQuery [Fis10] extension and we used the REST API of OpenCalais [Reu10d] to provide a lightweight alternative implementation for the library.

Furthermore we implemented some use cases to exemplify the usage of the library. Since GATE provides much more functionality and is extensible (unlike OpenCalais), we also defined an additional API on a different level of abstraction. This API is specific to the GATE implementation but allows easier implementation of some use cases.

1.4 Thesis Overview

In the following chapters we go through the different steps of our approach:

Chapter 2 presents different existing information extraction systems and investigates their functionality and suitability for being used for our XQuery library.

Chapter 3 specifies possible use cases for our library and defines the consequential requirements for the API. Finally, the API itself and a corresponding data model is being defined.

Chapter 4 first describes the structure of the implemented library components. Then it focuses on the components itself, including the implementation of an MXQuery extension to provide an interface to GATE and the implementation of the OpenCalais component which uses the OpenCalais REST interface.

Chapter 5 reviews the expressive power of the final library by revisiting the use cases of chapter 3 and presents the results of some basic performance measurements that show how the library scales and where the bottlenecks are.

Chapter 6 concludes the thesis and points out possible improvements.
Chapter 2

Overview of Information Extraction Systems

In this chapter we present some existing information extraction systems and comment on their features. We define a list of requirements as well as a reference text snippet to compare the different systems.

2.1 Requirements

2.1.1 Usability

The system should provide an API that allows to embed basic information extraction functionality into other software without too much effort. We use the example text to compare the required effort.

2.1.2 Good Out of the Box Performance

Using the system with default parameters (i.e. without additional configuration) should yield reasonable results. It should particularly recognize all person entities in the example text.

2.1.3 Coreference Resolution

The system should be able to establish some kind of connection between extracted information items. In particular, it should e.g. identify the entities Gordon Brown and Mr Brown as the same person in the example text. Using this text we also look at the capability of identifying pronouns in the same way, as another kind of connection between extracted items.

2.1.4 Support for Multiple Languages

While the systems that have been taken into consideration all work with English texts, we do not want to restrict ourselves to one language. Thus the system should also support other languages.
2.1.5 Extensibility

There should be a way to improve recognition of entities or information extraction performance in general. For example, one should be able to add entities to be recognized and to adjust the necessary “cogs in the wheel” for the recognition.

2.1.6 Openness

Since we intend to use the functionality of an information extraction system for our library, which we would like to publish as open source software, the license should be as open as possible regarding the usage of such a system. Ideally, the system is licensed in a way that does not restrict the licensing of the software that uses it (e.g., BSD [OSI10] or LGPL [FSF07b]).

2.1.7 Community and Active Development

Getting help in using and understanding the system should be easily possible, e.g., through bulletin boards (forums) or mailing lists. Furthermore, an active development of the system is desirable, such that continual improvement can be presumed.

2.2 Reference Text

We use the following text snippet to test for requirements 2.1.2 (Good Out of the Box Performance), 2.1.1 (Usability) and 2.1.3 (Coreference Resolution). For this example text we want to focus on the extraction of person entities, which is why we emphasized them below:

Gordon Brown was “marginalised” by Tony Blair throughout most of the build-up to the invasion Iraq, former international development secretary Clare Short has claimed. Clare Short insisted Mr Brown only came in in support of the military action after John Prescott patched up a reconciliation between him and Mr Blair. Ms Short, who gives evidence this week to the Iraq Inquiry, said Mr Brown feared that Tony Blair would use a quick victory over Saddam Hussein to strengthen his political position at home and remove him from the Treasury. She said that while Mr Brown – who was chancellor at the time – did not speak out in Cabinet against the war, he did not support it.

— Telegraph.co.uk [Tel10]

2.3 Reviews

2.3.1 MarkLogic Server

The software company MarkLogic [Mar10] offers an XML Server that contains an XQuery function module for Entity Enrichment. The module currently provides only one function (entity:enrich) that returns the “entity-enriched” node for
a given XML node (i.e. mentions of people, places\textsuperscript{1} etc. are surrounded by XML tags).

Putting our example text in a `<doc>` node and applying the `entity:enrich` function to it yields the following result (line breaks were added for readability):

```
<doc xmlns:e="http://marklogic.com/entity">
  <e:person>Gordon Brown</e:person>
  was "marginalised" by
  throughout most of the build-up to the invasion
  <e:gpe>Iraq</e:gpe>, former international development
  <e:person>secretary Clare Short</e:person>
  has claimed. Clare Short insisted
  <e:person>Mr Brown</e:person>
  only came in in support of the military action after
  <e:person>John Prescott</e:person>
  patched up a reconciliation between him and
  <e:person>Mr Blair</e:person>
  , who gives evidence this week to the
  <e:gpe>Iraq</e:gpe>
  Inquiry, said
  <e:person>Mr Brown</e:person>
  feared that
  <e:person>Tony Blair</e:person>
  would use a quick victory over
  <e:person>Saddam Hussein</e:person>
  to strengthen his political position at home and remove him from the
  Treasury. She said that while
  <e:person>Mr Brown</e:person>
  - who was chancellor at the time - did not speak out in Cabinet
  against the war, he did not support it.
  <e:person>Ms Short</e:person>
</doc>
```

First of all, the effort needed to get results is next to nothing. After importing the module, just one function call is needed, so requirement 2.1.1 (Usability) is clearly met. Unfortunately, there are some misses in people recognition. The second occurrence of Clare Short is not being recognized and the first mentioning of her is including her job title, which is not exactly what we want. Furthermore, there is not much information in the annotation tags – only the entity type – so there is no kind of coreference resolution either. While requirement 2.1.2 (Good Out of the Box Performance) is partly met, 2.1.3 (Coreference Resolution) apparently is not.

Besides English, the function also supports and auto-detects other languages, e.g. French, Spanish, German and Italian. However, there is no possibility to improve the recognition performance or enhance the module with more types of entities. Thus, requirement 2.1.4 (Support for Multiple Languages) is met but not 2.1.5 (Extensibility). Regarding requirement 2.1.6 (Openness), MarkLogic Server does need a license key and there are special licenses needed for additional functionality like the Entity Enrichment module. We would like to thank the sales stuff for giving us a temporary evaluation license for this thesis. There are mailing

\textsuperscript{1}The function differentiates between a location (e.g. Mount Everest) and a gpe (Geo-political entity, e.g. California because its boundaries were defined by a government).
lists and a MarkLogic Community Website where you can get help from. Further development is in progress as well, so requirement 2.1.7 (Community and Active Development) is met.

While the Entity Enrichment module certainly offers interesting possibilities, especially in connection with other MarkLogic modules like e.g. Geospatial Functions, it does not satisfy our needs that well.

### 2.3.2 MinorThird

William W. Cohen, professor in the Machine Learning Department at Carnegie Mellon University in Pittsburgh, has been involved in a number of research projects around information extraction – primarily using machine learning techniques – resulting in lots of publications [Coh10]. His – for our purpose most interesting project is MinorThird [Coh04], which stands for Methods for Identifying Names and Ontological Relations in Text using Heuristics for Inducing Regularities from Data and provides Java classes for storing and annotating text as well as for learning to extract entities and categorize text.

To run a test with our example text, we use the Java code shown below. First we create a text base in where we put our text, which is stored in the variable example_text. Then we create a new annotator based on the “MixupProgram” names.mixup – we explain Mixup programs later in that section – and use it to get annotated labels for our text base. After that we extract all annotations of type “Name” in our text and return them.

```java
String id = "myDoc";
BasicTextBase textBase = new BasicTextBase();
textBase.loadDocument(id, example_text);
MixupProgram mp = new MixupProgram(new File("names.mixup"));
MixupAnnotator ma = new MixupAnnotator(mp);
TextLabels ann_labels = ma.annotatedCopy(new BasicTextLabels(textBase));
Iterator<Span> it = ann_labels.getTypeSet("Name", id).iterator();
String result = "";
while (it.hasNext()) {
    result += it.next().asString() + "\n";
}
System.out.println(result);
```

The output is the following (note that the Span objects that we print asString() also contain position information, so an output similar to the one in subsection 2.3.1 would be possible too):

```
Hussein
Prescott
Clare
Blair
Brown
Clare
Gordon Brown
Brown
John Prescott
Brown
```
2.3. Reviews

The effort needed to get that result is reasonable while the result itself is rather disappointing. Sometimes only the first name is being recognized and sometimes only the last name. Additionally, all persons that have been recognized by their full name are recognized a second time by only their last name (e.g. John Prescott and Prescott). There is no coreference information stored in the Span objects either, so while requirement 2.1.1 (Usability) is more or less met, requirements 2.1.2 (Good Out of the Box Performance) and 2.1.3 (Coreference Resolution) clearly are not.

Requirement 2.1.4 (Support for Multiple Languages) is not met either. However, having a look at requirement 2.1.5 (Extensibility) puts a different complexion on the whole matter. MinorThird is extensible by design. In fact, its real strengths only become visible when the provided machine learning algorithms are used to create a powerful annotator. That is where Mixup programs come into play. Mixup is a special-purpose annotation language that comes with MinorThird and is somehow similar to regular expressions but working on tokens rather than on single characters. Generally, the idea is to create such a Mixup file by hand and – together with pre-annotated training data – let it improve itself using the learning methods provided by MinorThird.

In our test, we used a Mixup file/program that was already provided with the distribution of MinorThird. To get better results, we should have trained that program in advance. Still, doing it that way would result in a much greater effort, especially because it has to be done for each individual entity or relation that we want to have recognized. Thus, this would considerably affect requirement 2.1.1 (Usability), which is why we did not go for it.

Regarding the last two requirements, MinorThird is basically distributed under the BSD license\(^2\), so it perfectly fulfills 2.1.6 (Openness). There are no mailing lists for MinorThird and there is only sparse activity, looking at the project page. However in research MinorThird is quite well-known, so we consider 2.1.7 (Community and Active Development) as partly met.

2.3.3 OpenCalais

A pretty different – at least regarding the way of using it – kind of information extraction system (or rather service in this case) is OpenCalais [Reu10a], provided by the information company Thomson Reuters. OpenCalais offers information extraction as a web service providing different APIs, including a REST API [Reu10d].

There are multiple kind of outputs OpenCalais provides, the default (and most thorough) one being RDF. For the sake of readability we choose the Text/Simple

\(^2\)Note that MinorThird uses a number of third-party systems however, some of which are licensed under GPL [FSF07a]. In those cases the authors have informally agreed to a second release under LGPL for incorporation into MinorThird.
format with our example text (which is stored in the variable $example-text$). For the REST call in XQuery we use Zorba’s REST module [Zor09] (which is also built-in in MXQuery [Fis10]):

```xml
<CalaisSimpleOutputFormat>
  <Person count="5" relevance="0.786">Gordon Brown</Person>
  <Person count="4" relevance="0.771">Clare Short</Person>
  <Person count="3" relevance="0.246">Saddam Hussein</Person>
  <Person count="3" relevance="0.712">Tony Blair</Person>
  <Person count="2" relevance="0.329">John Prescott</Person>
  <Country count="1" relevance="0.360" normalized="Iraq">Iraq</Country>
  <Position count="1" relevance="0.134">chancellor</Position>
  <Topics>
    <Topic Taxonomy="Calais" Score="1.000">Politics</Topic>
    <Topic Taxonomy="Calais" Score="0.985">War_Conflict</Topic>
  </Topics>
</CalaisSimpleOutputFormat>
```

Preparing the REST call needs a few lines of code, but basically the usage is very easy. OpenCalais also recognized all persons, so requirements 2.1.1 (Usability) and 2.1.2 (Good Out of the Box Performance) are perfectly met. Although not obvious in the simple output format, 2.1.3 (Coreference Resolution) is met as well. In the RDF output format there is an entry for each occurrence of e.g. Gordon Brown and Mr Brown and also for personal pronouns referencing a person. And those entries are connected with each other, so by using the RDF output format all this information is accessible. A hint to coreference resolution in the simple output format is the count attribute.

However, coreference resolution with personal pronouns is not 100% perfect. In the example text there is only one occurrence each of Saddam Hussein and John Prescott whereas OpenCalais thinks there are 3 and 2 respectively. This is because it assigns the pronouns in the corresponding sentences to these persons instead of assigning them to Gordon Brown and Tony Blair respectively. But since these cases are quite difficult and we are mainly interested in coreference resolution in general, this is not a major problem.

OpenCalais supports English, French and Spanish, but they state that they continue to expand the list of supported languages. The API to OpenCalais is quite
limited and leaves no possibilities for extensions at all. Thus, requirement 2.1.4 (Support for Multiple Languages) is fairly met but 2.1.5 (Extensibility) is clearly not.

The non-extensibility also has to do with the fact that OpenCalais is closed, proprietary software. Nevertheless it is completely free to use\(^3\), even for commercial purpose, which is quite satisfying for requirement 2.1.6 (Openness). There is a Calais Community, i.e. they have forums with quite some activity where you can get help, and a lot of software using the service has already been written [Reu10b]. Further development of the service is also in full activity, so requirement 2.1.7 (Community and Active Development) is met as well.

### 2.3.4 GATE

Still being up-to-date today, the beginnings of the General Architecture for Text Engineering project date as far back as 1995 [CGW95]. Accordingly, there are a great many publications about and around GATE [GPb10]. Similarly, many people contributed to its development so that with all its plug-ins it now “weights” almost 300 MiB (the core with all libraries requiring roughly 50 MiB thereof).

Besides a GUI, GATE also provides a Java API to use its functionality. There are several kinds of XML formats for the output of the processed text. The most thorough one reveals GATE’s internal representation and is somewhat similar to the RDF format of OpenCalais. For our example text – again stored in the variable example_text – we use a more concise format where the annotations are directly printed in the original text.

```java
Gate.init();
// load controller
String annieRelPath = ANNIEConstants.PLUGIN_DIR + File.separator + ANNIEConstants.DEFAULT_FILE;
File annieFile = new File(Gate.getPluginsHome(), annieRelPath);
SerialAnalyserController controller = (SerialAnalyserController)
PersistenceManager.loadObjectFromFile(annieFile);
// create corpus and add document to it
String corpClass = "gate.corpora.CorpusImpl";
Corpus corpus = (Corpus) Factory.createResource(corpClass);
corpus.add(Factory.newDocument(example_text));
// annotate documents in corpus
controller.setCorpus(corpus);
controller.execute();
// print document as XML, including annotations of type "Person"
System.out.println(doc.toXml(doc.getAnnotations().get("Person"))));
```

ANNIE (A Nearly-New Information Extraction system) is the plug-in that provides the actual information extraction. After initializing GATE and loading the ANNIE controller we create a corpus where we add our document to. Then we execute ANNIE on that corpus and print the document in XML format, containing

---

\(^3\)The free to use OpenCalais service is restricted to 50’000 transactions per day at a maximum rate of four transactions per second (can be raised upon request). Furthermore it is noteworthy that OpenCalais keeps a copy of the metadata of each submitted content.
only the annotations labeled Person (doc.toXml() without parameters would output the thorough format mentioned above). This code yields the following output (line breaks were added for readability):

```
<Person gate:gateId="321" rule="PersonFull" gender="male" gate:matches="321;326;332;336" rule="PersonFinal">Gordon Brown</Person>

was "marginalised" by

 throughout most of the build-up to the invasion Iraq, former international development secretary

 has claimed.

 insisted

 only came in in support of the military action after

 patched up a reconciliation between him and

 , who gives evidence this week to the Iraq Inquiry, said

 feared that

 would use a quick victory over

 to strengthen his political position at home and remove him from the Treasury. She said that while

 - who was chancellor at the time - did not speak out in Cabinet against the war, he did not support it.
```

The effort needed to get the result is comparable to MinorThird, but the result is much better. Not only are all persons recognized correctly, but the gate:matches attributes also refer to annotations that relate to the same person. So while meeting requirement 2.1.1 (Usability) only more or less, GATE clearly fulfills requirements 2.1.2 (Good Out of the Box Performance) and 2.1.3 (Coreference Resolution).

Requirements 2.1.4 (Support for Multiple Languages) and 2.1.5 (Extensibility) are met as well, both due to the modular design of GATE and the numerous plug-ins available (some of them enabling other languages than English). Besides writing a plug-in, there is another mechanism (e.g. to annotate another type of

Note however, that we do not load an entity-specific annotator here but only limit the output to a specific entity type.

Coreference resolution including personal pronouns is supported by loading an additional component, which only takes one additional line of code. Interestingly the results are wrong for exactly the same pronouns in the example text as with OpenCalais.
2.4. Summary

An overview of which systems meet which requirements is given in Table 2.1. It is clearly visible that OpenCalais and GATE best fit our needs. OpenCalais lacks the possibility of being extensible and its license is not as free as GATE’s, but in turn OpenCalais is a bit easier to use.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>MarkLogic</th>
<th>MinorThird</th>
<th>OpenCalais</th>
<th>GATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.1 (Usability)</td>
<td>yes</td>
<td>partly</td>
<td>yes</td>
<td>partly</td>
</tr>
<tr>
<td>2.1.2 (Performance)</td>
<td>partly</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.3 (Coreference)</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.4 (Languages)</td>
<td>yes</td>
<td>no</td>
<td>partly*</td>
<td>partly**</td>
</tr>
<tr>
<td>2.1.5 (Extensibility)</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.6 (Openness)</td>
<td>no</td>
<td>yes</td>
<td>partly</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.7 (Community)</td>
<td>yes</td>
<td>partly</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

*) only a few yet  **) requires usage of plug-ins

Table 2.1: Summary of Systems and Requirements

We decide on using GATE as well as OpenCalais for our library. GATE best fits our needs and is especially interesting because of its openness and extensibility. Using GATE gives us a wide range of possibilities for text engineering in general – even though this thesis only looks at information extraction. On the other hand GATE is quite large (in terms of disk space) which is why we also choose OpenCalais as a lightweight alternative. The idea is that our library offers a choice to use either GATE (heavyweight, off-line and extensible) or OpenCalais (lightweight and on-line) as the underlaying system.
Chapter 3

API Design

Having chosen the candidates to be used in our library, the next step now is to define an API. But we do not want to base the API solely upon the available systems on the implementation side. It is probably even more important to take a look at the user side, which is why we present some use cases prior to finally defining the API.

3.1 Use Cases

3.1.1 Extraction of Person Entities

One important feature of information extraction systems is the recognition of persons in a text. As we have seen in chapter 2, this feature is much more powerful if it also identifies multiple occurrences of the same person, probably also taking personal pronouns into account. Therefore a possible use case is a summary of all distinct persons that occur in a text, containing information about the number of occurrences (including personal pronouns) and possible aliases.

If we take our example text from section 2.2, the expected output should look like the following:

```xml
<Persons>
  <Person count="7">
    <Name>Gordon Brown</Name>
    <AKA>Mr Brown</AKA>
  </Person>
  <Person count="4">
    <Name>Tony Blair</Name>
    <AKA>Mr Blair</AKA>
  </Person>
  <Person count="4">
    <Name>Clare Short</Name>
    <AKA>Ms Short</AKA>
  </Person>
  <Person count="1">
    <Name>John Prescott</Name>
  </Person>
  <Person count="1">
    <Name>Saddam Hussein</Name>
  </Person>
</Persons>
```
3.1.2 Entity Annotations

Besides extraction we are also interested in annotation of entities. That is the recognized entities should not be returned separately but rather stay in the input text marked by annotations. As an example consider the following text where we want to have the E-Mail addresses annotated:

Please contact John at j.doe+private_stuff@some.example.com. You could also write to john@d.oe, but he'll only check that inbox when he's@home. Jane can be reached@the following address: Jane#Smith@209.85.129.18 – although weird it's still a valid one.

The desired output when feeding this text into the library is something like this (without the line breaks, which have just been added for better readability):

```
1 Please contact John at
2 <EmailAddress>j.doe+private_stuff@some.example.com</EmailAddress>
3 . You could also write to
4 <EmailAddress>john@d.oe</EmailAddress>
5 , but he'll only check that inbox when he's@home. Jane can be reached@the
6 <EmailAddress>Jane#Smith@209.85.129.18</EmailAddress>
7 -- although weird it's still a valid one.
```

3.1.3 Extraction of Relations

Another quite useful feature is the recognition of relations between entities. If a program extracts a list of persons and a list of phone numbers from a document, there is by far more value if it also tells you which phone number belongs to which person. As an example we would like to extract all persons with their contact details out of the following text:

Dear Joe,

I think 1 800 123 4567 is the number of Robert Random. If he is unavailable you could also write to rr@example.info. He probably wanted to ask about the next meeting so you could also contact Sue Smith at 1 800 123 4568 or ss@eg.uk.

Best, Tom.

The expected result should look as follows:

```
1 <contact>
2  <Person>Robert Random</Person>
3  <EmailAddress>rr@example.info</EmailAddress>
4  <PhoneNumber>1 800 123 4567</PhoneNumber>
5  </contact>
6 <contact>
7  <Person>Sue Smith</Person>
8  <EmailAddress>ss@eg.uk</EmailAddress>
9  <PhoneNumber>1 800 123 4568</PhoneNumber>
10 </contact>
```

3.1.4 Other Examples

There are many other examples of use cases that are very similar to the ones described above. Because of their similarity, we do not present them separately but just give a short description of some use cases we get back on in Appendix B:
Wikipedia Links

In terms of the use case described in subsection 3.1.2 we may, for example, want to annotate the content of a web page with hyper links on entities. Such a hyper link could point to the corresponding Wikipedia [Wik10] article of the entity.

Twitter Topics

Regarding the use case of subsection 3.1.1 another example would be the extraction of entities that are discussed in a specific Twitter [Twt10] list. A Twitter list is a set of users “tweeting” about a certain topic. The goal would be to analyze such a list and, for example, extract persons or companies being discussed.

3.2 API Requirements

The design of the API should now consider both, the user side (based on the use cases) and the implementation side (based on the systems to be used). While the implementation side requires an interface that is as generic as possible, the use cases wish for as much functionality as possible. Joining these two aspects obviously leads to restrictions by finding a “lowest common denominator”. Thus the API should be designed according to the following specifications:

- Both systems, GATE and OpenCalais, support many different types of entities. However, the API would get unmanageable if there was a function for each single entity. Both systems do not allow to only extract certain entities anyway, thus there is no possible advantage in terms of performance either. Therefore the API should not provide entity-specific functions but rather let the caller pick the desired entities from the result. Nevertheless, this can be facilitated by providing a way to specify the desired entity types through function parameters.

- The API should provide functions that make it easy to implement the examples given in section 3.1. Thus, no complex or extensive algorithms – in particular no language specific processing – should be required for the implementation.

- The data model for the results returned by functions of the API should be coverable by both underlaying systems. However, there may exist a function allowing access to the genuine output of the underlaying system. In this way, the caller has a possibility to use implementation-specific data that would otherwise not be visible due to the generic approach.

- Given that GATE provides substantially more possibilities with different plug-ins, another API may be defined on a different level of abstraction – i.e. specific to GATE – that allows the inclusion of plug-ins for additional functionality.
3.3 API Definition

3.3.1 Overview

In the following, we call our library *xix*, as in XQuery Information Extraction. Figure 3.1 shows the basic structure of the different components. The user of the library can choose between the generic API or the GATE specific, lower-level API. For the generic API the underlaying system may be GATE or OpenCalais. This is determined statically, i.e. before runtime.

Figure 3.1: API design

3.3.2 Data Model

One decision that has to be made for the data model is whether to return “pure” entities or annotations within the input data. We choose to return annotations because it is easier to go from annotations to a list of entities than vice versa. For example, if we get the input annotated with entities of type **Person**, say in a variable $result$, we can get a list of persons simply by $result/Person$.

Figure 3.2 shows the types of annotations of our library. For lack of space, only the more complex types are expanded in the figure. All types have required *id* and *annId* attributes and optional arbitrary child nodes (for example HTML markup of the input data). The purpose of the attribute *annId* is to uniquely identify a single annotation while *id* is used for coreference among annotations that refer to the same real-world entity. The annotation of type **Person** has optional attributes *gender* and *mentionType* with possible values “male”/“female” and “name”/“pronoun” respectively. An **Organization** may have an attribute *type* with value “company” whereas the optional *type* attribute of **Location** can have one of the following values: “city”, “province”, “country” or “region”.

---

**Figure 3.2: Data model**
Figure 3.2: Types of annotations returned by the xir library
OpenCalais provides a huge amount of entities that are not supported through GATE by default (i.e. by using the ANNIE plug-in). Of the remaining entities we merged OpenCalais’ Country, City, Province and Region into one Location entity while keeping the specification as an optional type attribute (like GATE does). Similarly there is an optional type attribute if an Organization is actually a Company.

3.3.3 Functions

Due to the required generic nature we end up with basically only two different functions. One of them allows accessing the genuine output of the underlaying system while the other one returns a result according to the generic data model defined above. Yet, for the sake of convenience, the latter one is provided in four different “flavors”: On the one hand the caller may select which entities should occur in the output (including “all of them”). On the other hand the caller can choose between providing a URI or a node to specify the input data (which is offered with the “genuine output”-function as well). Another kind of providing input data would be a simple string. The reason why this is impractical is explained later on in subsection 4.2.1. Following are the signatures of the functions provided:

```xml
1 [: annotate the given $node with all types (Person, Organization etc.) :)]
2 xix:annotate($node as element())
3 as element()
4
5 [: annotate the given $node with the given types only :)]
6 xix:annotate($node as element(), $types as xs:string*)
7 as element()
8
9 [: annotate the document at $uri with all types :]
10 xix:annotate-doc($uri as xs:anyURI)
11 as document-node()
12
13 [: annotate the document at $uri with the given types only :]
14 xix:annotate-doc($uri as xs:anyURI, $types as xs:string*)
15 as document-node()
16
17 [: extract all information from the given $node (the output is highly implementation specific (e.g. an OpenCalais RDF or a GATE document))]
18)
19 xix:full-analysis($node as element())
20 as document-node()
21
22 [: extract all information from the document at $uri (the output is implementation-dependent (e.g. an OpenCalais RDF or a GATE document))]
23)
24 xix:full-analysis-doc($uri as xs:anyURI)
25 as document-node()
26
27
```

3.3.4 GATE-Specific API

While we restrict ourselves in the generic API of subsection 3.3.3 to reduce the impact of the underlaying system as much as possible, we allow accessing some
of the additional possibilities of GATE through a lower level API defined here.

Before the definition of the GATE-specific API, we explain some terms associated with those additional possibilities. As mentioned in subsection 2.3.4, GATE needs the ANNIE plug-in to actually do information extraction. A plug-in – among other things – consists of processing resources. Multiple processing resources form a so-called pipeline. For example, the default pipeline of the ANNIE plug-in consists of the following processing resources:

1. Document Reset PR
2. ANNIE English Tokeniser
3. ANNIE Gazetteer
4. ANNIE Sentence Splitter
5. ANNIE POS Tagger
6. ANNIE NE Transducer
7. ANNIE OrthoMatcher

It would go beyond the scope of this thesis to explain the purpose of all those processing resources. But the important thing to know is that each of them is processing the input in a certain way so to get the desired information extraction at the end of the pipeline. Furthermore, such a pipeline can be stored as a file – a gapp-file. The easiest way to create and store such pipelines manually is by using GATE Developer, the GUI for GATE.

As a side note, JAPE files (mentioned in subsection 2.3.4) are being used by some of the processing resources. Moreover, there is a processing resource – the “Jape Transducer” – that can be added to a pipeline and with which one can incorporate custom JAPE files.

Having stored a pipeline as a gapp-file, it can be passed to the annotations function defined below to process the input with the desired system. The annieAnnotations function however loads the gapp-file bundled with the ANNIE plug-in, which defines the default ANNIE pipeline. Additionally, that function loads the “ANNIE Pronominal Coreferencer” processing resource for coreference resolution of personal pronouns (because this one is not defined in the default ANNIE pipeline).

So the GATE-specific API consists of the following functions:

```xml
1  (: annotate $node using the provided gapp-file $gapp :)
2  gate:annotations($gapp as xs:string, $node as element())
3      [, $types as xs:string*]}
4      as element()}
5
6  (: annotate the document at $uri using the provided gapp-file $gapp :)
7  gate:annotationsFromFile($gapp as xs:string, $uri as xs:anyURI
8      [, $types as xs:string*]}
9      as document-node{)
10```
For the sake of brevity we used square brackets to indicate parameters that occur in further versions of the otherwise identical function. If $\text{types}$ is provided, the output only contains annotations of the given types. Otherwise a “GATE Document”, i.e. an XML document containing GATE’s full analysis, is returned. Being GATE-specific, all these functions return the data untransformed. Details on the GATE format can be read in GATE’s User Guide [GUs10] (especially in Chapter 5).
Chapter 4

Implementation

4.1 Overview

As mentioned in previous chapters, we want to provide two alternative implementations of our library: one with GATE and one with OpenCalais as the underlaying information extraction system. In the following sections of this chapter we will explain the two different implementations. Besides a shared module containing constants (not shown in the figures), there are basically three XQuery modules involved. One for the GATE implementation, named xix-gate, one for the OpenCalais implementation – xix-calais – and the xix module that imports either of the implementation modules and provides the generic API defined in subsection 3.3.3. Note that the choice of the implementation is static, i.e. can not be changed during runtime.

4.2 GATE Implementation

Figure 4.1 illustrates the GATE implementation of our library. As described in section 3.3 the generic API and the GATE-specific API can be used. The implementation consists of three parts: the XQuery part, the MXQuery extension part and the GATE abstraction part. These are explained “bottom up” in the following subsections, after mentioning some particulars of GATE’s Java API.

4.2.1 GATE’s Java API

An example of the usage of GATE’s API is given in chapter 2. We show the first few lines of the output resulting from that example here again, to discuss some details concerning the XML output of GATE:

1 <Person gate:gateId="321" rule="PersonFull" gender="male" gate:matches="321;326;332;336" rule="PersonFinal">Gordon Brown</Person>
2 was &apos;&apos;marginalised&apos;&apos; by
3 <Person gate:gateId="322" rule="PersonFull" gender="male" gate:matches="322;328;333" rule="PersonFinal">Tony Blair</Person>

1Whenever we mention XQuery in this chapter we refer to XQuery 1.1 according to the W3C Working Draft 15 December 2009 [W3C09]. Hence any XQuery engine that executes that code has to support XQuery 1.1.
Figure 4.1: GATE Implementation
4.2. GATE Implementation

throughout most of the build-up to the invasion Iraq, former international development secretary Clare Short has claimed.

We notice the following:

1. There is no root element embedding the output.

2. The annotations (Person elements in this case) are associated with the default element namespace, which is not specified though.

3. Some attributes are associated with the prefix gate. However a namespace declaration for the prefix gate is missing.

There seems to be no way of “tuning” GATE to fix the output accordingly in this particular case. While the first two issues are actually tolerable in an XQuery environment, the third is not. Fortunately there is a workaround to overcome at least the first and third issue: putting the input text in a parent (root) element before passing it to GATE.

That this addresses the first issue is obvious. Regarding the third issue, GATE puts the namespace declaration for the prefix gate into the root element of the input if it detects such a root element.

Now, there are basically two approaches:

1. Add a root element to the retrieved string (or other “non-rooted” input).

2. Only accept “rooted” input.

We decided on requiring an element node (“rooted” input) of the caller for the following reasons: Not only is it somehow inappropriate (in this context) to return an element node if the caller provides e.g. a string, but it is also awkward constructing an artificial node that unnecessarily exposes a workaround. Furthermore it is easy for the caller to put “non-rooted” data into an element node of his own choice.

On that account the functions in subsection 3.3.3 only accept an element node as the parameter containing the data to be annotated (or, alternatively, a URI pointing to a well-formed XML document that, by definition, also does have a root element).

4.2.2 EasyGate – An Abstraction of GATE’s Java API

Although GATE’s Java API is reasonably easy we decided to build an additional level of abstraction to facilitate certain operations, reduce the number of non-basic Java classes involved and to generally decouple the MXQuery extension from the GATE API. Hence changes in the GATE API essentially only affect the EasyGate class while changes in the way MXQuery extensions are made only affect the MXQuery extension.

The following methods are provided by our Java class EasyGate:
All these methods as well as the class itself are static. One reason for this is that loading a controller takes quite some time and we therefore only allow working with one instance so that when processing multiple documents we are able to make sure the controller is only loaded once. We are aware that there are better solutions and the intention is to revise this in future work.

The first four methods can be used for setting up the desired system while the fifth method – `reset()` – will undo these settings. Then, there are methods to create documents and corpora (collections of documents) and a method to execute the controller on a corpus (a controller cannot be executed on a document directly). Finally, with the last two methods one can either get a document with annotations of certain types or an XML document containing all information extracted by the controller.
4.2. GATE Implementation

As an example of using the EasyGate class we present the code needed to achieve the same result as in chapter 2 (assuming the example text is stored in the string `example_text`):

1. `EasyGate.initialize();`
2. `EasyGate.loadController();`
3. `EasyGate.createCorpus("myCorpus");`
4. `EasyGate.createDocument("myDocument", example_text);`
5. `EasyGate.addDocumentToCorpus("myCorpus", "myDocument");`
6. `EasyGate.executeWithCorpus("myCorpus");`
7. `HashSet<String> types = new HashSet<String>();`
8. `types.add("Person");`
9. `System.out.println(EasyGate.getAnnotatedDoc("myDocument", types));`

Although about the same amount of code is needed, the user of EasyGate does not have to care about GATE-specific Java classes like `SerialAnalyserController` or `Corpus`. Additionally, EasyGate transparently takes care of reusing an already loaded controller.

4.2.3 MXQuery Extension

This section addresses the actual core for making GATE functionality available to XQuery. For that purpose, an adapter from Java to XQuery is needed. MXQuery is an XQuery engine written in Java and provides a framework to implement custom extension functions. On the one hand, a so-called function gallery – an XML document – has to be created, specifying what XQuery functions the extension contains. On the other hand, the actual implementation of these functions is done by writing an MXQuery iterator class.

The XQuery functions that are provided by this iterator (and specified in the function gallery) exactly represent the lower-level / GATE-specific API as defined in subsection 3.3.4. The iterator makes use of the methods provided by the EasyGate class. Since GATE (and consequently EasyGate) only takes strings as input and returns strings as output, the iterator also takes care of serializing the input and deserializing the output. Explaining the iterator in more detail would be more or less all about how to write an MXQuery iterator, which would go beyond the scope of this thesis.

Note that, unlike the other parts, this part of the library is specific to the MXQuery engine. If using another engine, one would have to implement a corresponding adapter to provide the GATE-specific API using the EasyGate methods.

4.2.4 XQuery Module

Finally, the `xix-gate` module connects the generic API provided in the `xix` module with the MXQuery extension. It uses the functions of the GATE-specific API provided by the MXQuery extension and transforms the retrieved result according to the generic data model defined in subsection 3.3.2.
4.3 OpenCalais Implementation

As with the GATE implementation we discuss the implementation for OpenCalais from the bottom up - though there are much less different components in this case anyway, as shown in Figure 4.2. In the OpenCalais implementation everything is written in XQuery and is thus independent of the XQuery engine.

![Figure 4.2: OpenCalais Implementation](image)

4.3.1 OpenCalais REST API

The REST API of OpenCalais is about as simple as it gets. There is a Web Service URL and by using HTTP GET or HTTP POST the parameters `licenseID`, `content` and `paramsXML` have to be passed. The value for the `licenseID` is the API key you get when you register with OpenCalais. With `content` you pass the data you want to have analyzed and `paramsXML` specifies some XML-formatted parameters [Reu10c] indicating how OpenCalais should process the data. An example of using the REST API in XQuery can be reread in chapter 2.

4.3.2 XQuery Module

The `xix-calais` module contains the REST calls to OpenCalais and transforms the response according to the generic data model defined in subsection 3.3.2. Unlike with GATE, we do not have a choice of annotated input data as a result but have to process OpenCalais’ RDF format. So basically, for each entry of interest in the RDF we have to get the position and construct the corresponding annotation in the original data.
4.3. OpenCalais Implementation

During the implementation of this transformation, we met some problems concerning personal pronouns. The RDF entry of such a pronoun is not distinguishable from an entry of a name referring to the same person. Furthermore, there is also no information about the gender of a person, so using the OpenCalais implementation one cannot take advantage of the corresponding optional attributes of the Person type in our data model. Nevertheless, we reckon that this information will be available from OpenCalais in the future so we did not further restrict the data model.
Chapter 5

Evaluation

This chapter discusses the expressive power of the final library on the basis of the implementations of the example use cases given in section 3.1. Furthermore, the results of some performance measurements are presented to get an idea of the scalability of the whole library.

5.1 Functionality

As we know from the previous chapters, the xix library does not provide a vast variety of different functions due to its generic nature. However, the purpose of an information extraction library is essentially only one function that extracts or annotates all desired entities or relations (though there are several ways to achieve this goal). The result can then be used for further processing. This is demonstrated in the following, by implementing the examples of the use cases Extraction of Person Entities, Entity Annotations and Extraction of Relations.

5.1.1 Person Extraction

Below is the code needed to get the expected result for the example of the Extraction of Person Entities use case. $example refers to an element node containing the example text and $xixs:PERSON is the constant telling the annotate function that we only want annotations of type Person in the response.

```xml
<Persons>
  let $annDoc := xix:annotate($example, $xixs:PERSON)
  let $persons := $annDoc//Person
  let $longestOccurences :=
    for $o in $persons
      let $id := $o/@id
      group by $id
      return (: the (first) occurrence with maximum string-length :) 
        ($o[fn:string-length(.) eq 
          fn:max( for $s in fn:data($o) 
            fn:string-length($s))]
          return fn:string-length($s))
    )[[1]
  for $lo in $longestOccurences
    let $count := fn:count($persons[@id eq $lo/@id])
    let $akaValues :=
      for $p in $persons[@id eq $lo/@id and text() ne $lo/text()]
        where fn:not($p/@mentionType) or $p/@mentionType ne "pronoun"
```

return $p$
return
<Person count="{$count}">
  <Name>{$lo/text()}</Name>
  { for $a in fn:distinct-values($akaValues)
    return <AKA>{$a}</AKA> }
</Person>
</Persons>

After getting all annotations of type Person we store the longest occurrence of each distinct person in $longestOccurrences. Then, for each item in that variable (i.e. again for each distinct person), we store all other distinct occurrences of that person – except occurrences that are personal pronouns – in $akaValues and return them together with the longest occurrence and the total number of occurrences (including personal pronoun occurrences).

Although some effort is needed to get exactly the desired output, it is all about processing and transforming the result of the annotate function. Thus this use case can be marked as “covered” by the library.

### 5.1.2 E-Mail Address Annotation

Implementing the example for the Entity Annotations use case is effortless. Since our library outputs recognized entities as annotations already, all we have to do is call the annotate function with the corresponding parameter for the type:

```
xix:annotate($example, $xixs:EMAIL-ADDRESS)
```

### 5.1.3 Contact Extraction

Regarding the example of the Extraction of Relations use case, there is not much information given about relationships in the result returned by the annotate function. Although the data model defined in subsection 3.3.2 would allow to, for example, associate a person entity with a phone number entity through the id attribute, this is not implemented yet.

The reason is that for GATE there is no plug-in so far that would recognize such a relationship. Furthermore, while OpenCalais does recognize relationships – and they would also be accessible using the xix:full-analysis function of our library – it performs not very good in that regard.

In fact, OpenCalais does not (yet) perform much better than the following implementation which uses a very naive algorithm that associates phone numbers and e-mail addresses with a person if they appear after the person occurrence:

```
let $a := xix:annotate($example,
  ($xixs:PERSON, $xixs:PHONE-NUMBER, $xixs:EMAIL-ADDRESS))
let $longestOccurrences :=
  for $o in $a/Person
    let $id := $o/@id
  group by $id
  return (: the (first) occurrence with max. string-length :)
```

For example, the text “Mr Obama is also known as Barack Hussein Obama” contains one distinct person with *Barack Hussein Obama* as the longest occurrence of that person.1

---

1 For example, the text ‘Mr Obama is also known as Barack Hussein Obama’ contains one distinct person with *Barack Hussein Obama* as the longest occurrence of that person.
(40)\[fn:string-length(.\) eq\]
\[fn:max( for \$s in fn:data($o)\]
\[return fn:string-length($s))\]
\]}[1]

for $lo in $longestOccurences
let $id := $lo/@id
return
for tumbling window $w in $a/element()[@id]
start $s next $snext
when $s/@id eq $id and fn:local-name($snext) =
($xixs:PHONE-NUMBER, $xixs:EMAIL-ADDRESS)
end $s next $snext
when fn:local-name($snext) eq $xixs:PERSON
return
("\n",
copy $c := <contact>{$w}</contact>
modify replace node $c//Person
with $longestOccurences[@id eq $c//Person/@id]
return $c)
)

Here a tumbling window [CFF+07] is used to gather phone number and e-mail address entities that follow an occurrence of a person entity. There are also some lines of code (3–11) that take care of getting the longest occurrence of a person, to be used in the contact summary afterwards.

Predictably, this implementation does not perform well on our example text:

```xml
<contact>
  <Person gender="male" mentionType="name"
    id="157" annId="157">Joe</Person>
</contact>
<PhoneNumber id="158" annId="158">800 123 4567</PhoneNumber>

<contact>
  <Person gender="male" mentionType="name"
    id="159;228;229" annId="159">Robert Random</Person>
  <EmailAddress id="160" annId="160">rr@example.info.</EmailAddress>
</contact>

<contact>
  <Person gender="female" mentionType="name"
    id="161" annId="161">Sue Smith</Person>
  <PhoneNumber id="162" annId="162">800 123 4568</PhoneNumber>
  <EmailAddress id="163" annId="163">ss@eg.uk.</EmailAddress>
</contact>
```

While the missing “1” of the phone numbers is a GATE-issue and can be avoided, for example, by using the xix-calais module instead, the first phone number is associated with the wrong person (since it is mentioned before the person it belongs to, which our naive algorithm does not consider).

To address this issue, there are several approaches. One would be to improve the implementation of our library. However, this would disagree with the generic API as long as relationship extraction is not available in GATE.

Nevertheless, one could use the xix:full-analysis function in connection with the OpenCalais implementation of our library to make use of the relationship information contained in OpenCalais’ RDF. The obvious drawback of that approach is that one has to fiddle with the whole RDF structure.

Using the GATE-specific API, a possible solution would be to write one’s own
GATE plug-in or JAPE file doing the job. Then, with GATE Developer, a pipeline can be created using the corresponding processing resource(s). Having stored that pipeline as a gapp-file, it can be used with the function gate:annotations of the xix library. However, at the time of writing this thesis, we have not had the chance to try this out.

All in all, with the current state of the library, there are three approaches:

1. Writing a GATE plug-in and using the gate:annotations function of the GATE-specific API
2. Using the xix:full-analysis function in connection with the OpenCalais implementation and transforming the RDF accordingly
3. Still using the generic xix:annotate function but implementing a better algorithm (probably using language-specific techniques) in XQuery

5.1.4 Other

The implementations of the other examples mentioned in section 3.1 are described in Appendix B.

5.2 Performance

In this section, we take a look at how long it takes to execute code that uses the xix library. The results of a few experiments are shown, delivering insight into the scalability of the library. We also took measurements for the use case implementations presented in section 5.1 but do not present those results separately because the differences (caused by a few lines of additional XQuery code) turned out to be perfectly negligible.

All measurements have been conducted using the XQuery engine MXQuery on a 1.3 GHz dual core system with 4 GiB RAM, a SATA solid state drive and a 64-bit operating system. If not stated otherwise, time measurements have been taken using the “-t” option of MXQuery, which measures the execution time of the query. Furthermore, the GATE implementation is used, unless stated otherwise.

5.2.1 Increasing the Document Size

For this experiment we used the public data dump of Stack Overflow [SO09] to create documents with different sizes. We altered the data in a way that the actual contents are stored as text nodes rather than attribute values so that annotations are possible at all. The documents created for the experiment are of the sizes 1, 2, 4, 8, 10, 20, 40, 80, 100, 200, 400 and 800 KiB.

Figure 5.1 shows how the execution time increases with increasing file size when calling the functions xix:full-analysis-doc or xix:annotate-doc (with one parameter) respectively. Five measurements were made for each document. The median values for each group of five measurements are connected through straight lines in the graph.
5.2. Performance

While both functions show linear scaling, the time increase is slightly less for the function `annotate-doc` compared to `full-analysis-doc`. This makes sense since the former returns only those types of annotations defined in the `xix` data model while the latter contains information about anything GATE finds in the document (e.g. `Sentence`, `Token` etc.), which generally results in a larger output especially for larger inputs.

Using the `annotate-doc` function, Figure 5.2 compares the GATE implementation with the OpenCalais implementation. Since OpenCalais only allows documents below 100 KB, the four biggest documents have to be omitted.

The variance of the five measurements for each document is much higher using the OpenCalais implementation. The network usage and the fact that a whole RDF document is transferred every time becomes clearly noticeable. This graph also demonstrates that the advantage of the otherwise faster alternative vanishes for documents greater than about 10 KiB. Regarding the GATE implementation, we will try to find the bottleneck showing up with smaller documents as follows.

**Detailed Analysis for the GATE Implementation**

Due to the fact that the GATE implementation, among other things, consists of an MXQuery iterator written in Java, we can take a closer look at which step takes how long to process. MXQuery is shipped with a Java class `PerfTools` that can be used to easily get performance measurements.
Figure 5.2: Comparing the GATE with the OpenCalais implementation

Figure 5.3 illustrates a breakdown of CPU time needed for each operation; starting with the reading of the function parameters and ending with the parsing of the result returned by GATE. Each curve represents a different document size. Unlike before, we hid the single measurements in this graph and only show the median values, since the variance of the measurements is very small with the GATE implementation anyway.

Note that the CPU time in Figure 5.3 is drawn on a log scale, so the differences between different file sizes in the 6th and 7th operation are much higher than the differences in the 4th and 5th operation. Furthermore we can see that the 3rd operation (loading the controller) is taking quite some time regardless of the file size of the document. It seems that this is the main bottleneck that becomes noticeable with smaller documents.

Figures 5.4 and 5.5 – where each curve represents an operation – support this assumption. The latter is a close-up of the former. Operations 2 and 3 (initializing GATE and loading the controller) always take around 0.8 and 3.5 seconds respectively. So for documents smaller than about 40 KiB, loading the controller is the most costly operation.

5.2.2 Increasing the Number of Documents

To investigate the behavior for an increasing number of documents, we made use of the same data as before. This time we created one thousand files of about 4 KiB each. The measurements were taken for 1, 2, 4, 8, 10, 20, 40, 80, 100, 200, 400 and 800 files.
Figure 5.3: Breakdown of CPU time into different operations (each curve is representing a different document size)
Figure 5.4: Operation costs for increasing file size

Figure 5.5: Operation costs, zoomed in
5.2. Performance

Since the class EasyGate described in subsection 4.2.2 is implemented in a way that an already loaded controller is not being loaded again, not all costs will, for example, double when doubling the amount of documents. This is visible in Figure 5.6 which shows the cumulative CPU times needed for the different kind of operations in relation to the amount of documents processed (similarly zoomed in as in Figure 5.5 – the full version looks similar to Figure 5.4).

![Figure 5.6: Operation costs for increasing amount of files, close-up](image)

Operation 6 increases linearly as expected. The slope is larger than the one of Figure 5.5 because in this graph, 10 documents – for example – correspond to 40 KiB of data while in the other graph it was 10 KiB, so the two graphs cannot be compared directly. However, we see that initializing GATE and loading the controller does not increase with the number of documents (due to the way EasyGate is implemented). A slight increase can be observed for the creation of a document in GATE. That makes perfectly sense for an increasing amount of documents. Nevertheless, the increase is extremely small compared to the cost of loading the controller (even though that operation is executed only once). For the other operations the increase is absolutely negligible.

We also compared the GATE with the OpenCalais implementation for an increasing number of documents. Unfortunately, we did not get very useful results to display in a graph due to the network usage of the OpenCalais implementation, which causes too extreme variance. But generally the OpenCalais implementation took longer to execute for more than one document – sometimes only a bit longer and sometimes much longer, it was not possible to see a relation to the number of documents.
Chapter 6

Conclusion

6.1 Summary

This thesis presented the construction of an information extraction library for XQuery. The implementation of this library makes use of existing information extraction systems while providing a generic API independent of the underlaying system. Nevertheless, we introduced an API on a lower level, so that additional functionality provided by GATE can be utilized.

Furthermore, we described how well the library can be used to implement the previously defined use cases. One of those use cases (Extraction of Relations) was not easy to implement and revealed some limitations of the current implementation. To address these limitations additional effort is needed, e.g. by writing a plug-in or JAPE file for GATE and using our library’s lower level API with it. Unfortunately, for lack of time, we could not present how to do that in this thesis.

Finally, performance measurements gave a picture of the scalability of the whole library and we could approximately evaluate when using the GATE implementation is recommended and when the OpenCalais implementation is more advantageous. We also gained insight into the bottlenecks of GATE in different setups.

6.2 Future Work

The current implementation of the \textit{xir} library has some limitations, some of which have already been denoted before. This section presents possible improvements regarding those and other limitations.

6.2.1 Facilitate Relationship Extraction

An approach to facilitate relationship extraction would be on the GATE-specific API. GATE has a plug-in for ontologies. By extending the GATE-specific API by a few additional functions, using such ontologies and thus getting relationship extraction could be simplified.
6.2.2 Revise the EasyGate Class

The way to prevent an already loaded controller from being loaded again by just defining everything in EasyGate as static may be not the best solution. For a more elegant one, it is probably a good idea to further investigate GATE itself (e.g. what can be shared, what can be executed multiple times, what has to be reinitialized). One possibility may be to allow multiple shared GATE instances (instead of just one), one for each different controller (i.e. pipeline) needed. The current implementation performs badly if e.g. two different controllers are used alternately.

6.2.3 Provide Higher-Level Functions

Another possible improvement is to provide some functions on top of the xix library that further facilitate the implementation of certain use cases. A lot of similar use cases can be generalized, thus providing functions for such a class of use cases would come in handy.

6.2.4 Support for Multiple Documents

Our library only supports functions for one document as input. Although there is a workaround by putting multiple documents in one surrounding element, it would be more convenient if the functions accept a sequence of elements or a sequence of URIs.

6.2.5 Evaluation of the Information Extraction Quality

In chapter 5 we primarily looked into the functionality and performance of the library. Another interesting criterion to investigate is the quality of the information extraction. One may evaluate if there are cases where the encapsulation causes quality losses compared to using GATE or OpenCalais directly.

6.2.6 Classification of Information Extraction Systems

It would be interesting to find criteria to classify the power of an information extraction system in general. There may be certain kinds of use cases that cannot be covered by certain kinds of information extraction systems. Like Turing completeness for programming languages, one may similarly determine some completeness or classification criteria for information extraction systems.
Appendix A

How to Use the Library

A.1 Download and Installation

The xix library is published as open source software on SourceForge. It is available from http://sourceforge.net/projects/xixlib/ as a ZIP file. To use the GATE implementation you also need GATE and MXQuery, both of which are available from the same location. The library should also work with newer versions of GATE and MXQuery, but it was only tested with the versions provided on that download page.

After downloading, unzip the library. This will create a folder named xix. Put MXQuery (mxquery.jar) and unzip GATE into this folder. There should now be a subfolder named gate. After this three steps the files and folders should be structured like shown below:

```
+ data
| - gate
|   | - bin
|   |   | gate.jar
|   |   | log4j.properties
|   | + lib
|   | + plugins
|   | gate.xml
| - lib
| xix.xq
| xix-calais.xq
| xix-gate.xq
| xix-strings.xq
| - xq
|   | examples.xq
|   | ...
| GateExtensions.xml
| mxquery.jar
| xix.bat
| xix.jar
| xix.sh
```

The folders data and xq contain examples, they are not essential for the library to work. The lib folder contains the XQuery part of the library while GateExtensions.xml and xix.jar form the MXQuery extension for the
GATE implementation. The scripts \texttt{xix.bat} and \texttt{xix.sh} are just for convenience to spare you typing the long command line.

\subsection*{A.2 Usage}

To use the library with the GATE implementation or with MXQuery as the XQuery engine, a Java installation is required. The easiest way is to write your XQuery code in a file – say \texttt{myApp.xq} – and store that in the \texttt{xix} folder. Then, in a command line, change to the \texttt{xix} directory and type the following line (in a UNIX environment type colons instead of semicolons in the classpath):

\begin{verbatim}
java -Dgate.home=gate -Djava.ext.dirs=gate/lib/ext/guk.jar -Dgate.user.config=gate-user.xml -Dgate.user.session=gate.session -classpath xix.jar;gate/bin/gate.jar;gate/bin;gate/lib/*;mxquery.jar ch.ethz.mxquery.cmdline.MXQuery -11m -um -func GateExtensions.xml -f xix/myApp.xq
\end{verbatim}

First you have to specify some variables for GATE. They are not all essential but it is recommended to specify them so that GATE does not make wrong guesses for their values. Next, you need to add the implementation of the MXQuery extension functions (\texttt{xix.jar}) to the classpath as well as GATE, GATE’s libraries, the folder where GATE’s \texttt{log4j.properties} can be found and MXQuery. Finally, we call MXQuery with options for XQuery 1.1, XQuery Update Facility, specify the function gallery of our MXQuery extension and lastly the file with the “main” XQuery code.

When processing large documents it may also be required to increase the maximum heap space for Java by putting e.g. \texttt{-Xmx1g} right after \texttt{java}. The scripts provided with the library contain exactly the command line above (including the maximum heap space option). As a minimal example consider the file \texttt{myApp.xq} containing the following code:

\begin{verbatim}
1 import module namespace xix = "http://xixlib.sf.net/xix" at "lib/xix.xq";
2 xix:annotate(<text>This is John Doe. He works for IBM.</text>)
\end{verbatim}

Then typing \texttt{.\textasciitilde\textbackslash xix.bat xix\textasciitilde myApp.xq} on the command line would yield the following result:

\begin{verbatim}
1 <text xmlns:gate="http://www.gate.ac.uk">This is <Person gender="male" mentionType="name" id="29;41" annId="29">John Doe</Person>. <Person mentionType="pronoun" id="29;41" annId="41">He</Person> works for <Organization type="company" id="30" annId="30">IBM</Organization>.</text>
\end{verbatim}

For more examples have a look at the file \texttt{examples.xq} and the related modules in the \texttt{xix} folder.
Appendix B

Other Use Case Examples

B.1 Wikipedia Links

An example usage of our library is turning recognized entities like Person, Organization or Location into hyper-links that point to the corresponding Wikipedia article. For demonstration we took the mobile version of a news article. Figure B.1 shows a section of the news article in its original version and in the version returned by the XQuery code below.

In this excerpt there is one hyper-link each to the Wikipedia article of Los Angeles, Reuters and CNN and four hyper-links to the article of Larry King. Additionally there are tooltips on the hyper-links, e.g. if the mouse cursor hovers over the word “his” in the excerpt, there is a tooltip showing “Larry King”. The complete code to achieve this using our library is the following:

```
import module namespace xix = "http://xixlib.sf.net/xix" at "lib/xix.xq";

import module namespace util = "http://www.zorba-xquery.com/zorba/util-functions";

declare variable $local:types as xs:string* :=
("Person", "Organization", "Location");
```
We use the Zorba `util` function [Zor10] `tdoc` that makes sure the document is well-formed XML (by using Tidy [Rag10]). The root element of the tidy document is then passed to the `xix:annotate` function of our library returning the news article with annotations of types Person, Organization and Location.

We then modify a copy thereof. First we group all annotations of interest by the `id` attribute and get the longest occurrence for each group, i.e. for each distinct real world entity. We then loop through all annotations, construct the hyperlink to the Wikipedia article for the current annotation and finally replace the annotation with a HTML hyper-link element unless the occurrence of that entity is already surrounded by a hyper-link.

### B.2 Twitter Topics

A use case similar to the one in subsection 3.1.1 is the extraction of topics (e.g. persons or organizations) discussed in a Twitter list. Twitter users can define lists of authors they want to follow. Those lists can be public so other users can follow the same list of authors. Since Twitter provides a REST API [Twt09] to access such lists, the implementation of this use case is quite straightforward. Here is the complete code, analyzing the list “bigdata-analytics” of user “alisohani”:

```sql
import module namespace xix = "http://xixlib.sf.net/xix" at "lib/xix.xq";
import module namespace rest = "http://www.zorba-xquery.com/zorba/rest-functions";
```
Let $user := "alisohani"$
let $list := "bigdata-analytics"
let $url := fn:concat("http://api.twitter.com/1/", $user, "/lists", $list, "/statuses.xml?per_page=200")
let $response := rest:get($url)
let $t := [texts]
fn:string-join($response//text/text(), "\n")
$topics := $annDoc//*[fn:local-name(.) = $types]
$longestOccurences :=
for $o in $topics
let $id := $o/@id
group by $id
return ($o[fn:string-length(.) eq
fn:max(for $s in fn:data($o)
return fn:string-length($s))])[1]
for $lo in $longestOccurences
let $count := fn:count($topics[@id eq $lo/@id])
where $count > 1
return element [fn:local-name($lo)] {
attribute ["count"] {($count), $lo/text()}
}

We use Zorba's REST module for the REST call again. Next, we extract the actual text users wrote and put them all in a $texts$ element. This is what we pass to the $xix:annotate$ function of our library, together with the types of annotations we want to have in the result. We then store the longest occurrences of annotations referring to the same real world entity, as already seen in other use case implementations before.

Finally we count the number of occurrences of each distinct entity and return an element containing this information together with the type and the longest occurrence of the entity. Note that additionally – for the sake of brevity – we only output entities that occur more than once.

The resulting output is as follows:

We use Zorba's REST module for the REST call again. Next, we extract the actual text users wrote and put them all in a $texts$ element. This is what we pass to the $xix:annotate$ function of our library, together with the types of annotations we want to have in the result. We then store the longest occurrences of annotations referring to the same real world entity, as already seen in other use case implementations before.

Finally we count the number of occurrences of each distinct entity and return an element containing this information together with the type and the longest occurrence of the entity. Note that additionally – for the sake of brevity – we only output entities that occur more than once.

The resulting output is as follows:
There are some obvious false positives, so there is room for improvement concerning the information extraction system (GATE with ANNIE in this case). For better results, one may take advantage of the ability to use plug-ins with the GATE-specific API and, for example, define additional rules in a JAPE file for this specific Twitter list.
Bibliography


[Fis10] Peter Fischer. MXQuery – A lightweight, full-featured XQuery Engine.

[FSF07a] FSF - Free Software Foundation, Inc. GNU General Public License.

[FSF07b] FSF - Free Software Foundation, Inc. GNU Lesser General Public License.


[Twt09] Twitter REST API Method: GET list statuses.  

[Twt10] Twitter – a social networking and microblogging service.  


[Zor09] Zorba – The XQuery Processor. Description of the REST module.  

[Zor10] Zorba – The XQuery Processor. Description of the “Zorba util” functions.  