Universität Zürich
Institut für Informatik

Annotating Political Entities in Newspaper Articles: A COSA Implementation Using Finite-state Devices

Lizentiatsarbeit der Philosophischen Fakultät der Universität Zürich,
betreut von Alexandra Bünzli, lic. phil. und Prof. Dr. Michael Hess

Lukas Rieder
Untere Halde 15
5400 Baden

lukas.rieder@email.ch

September 2008

Contents

1 Introduction

A collaboration between the Institute of Computational Linguistics and the Department of Comparative Politics, both at the University of Zurich, has presented the opportunity for this master's thesis. The joint project, which is dubbed "COSA" for "core sentence annotation", aims at automating parts of a textual annotation process that is used to extract relations between relevant political entities from newspaper articles.[1] These entities are actors on the one hand (e.g. "Tony Blair", "Democrats") and topics on the other hand (e.g. "child care"). The relations between them are expressed in *subject-predicate-object* triples, a representation inspired by Kleinnijenhuis, de Ridder, and Rietberg (1997) and called *nuclear sentence* or *core sentence*. The data thus yielded is open to all sorts of analysis. Current efforts are focused on the *NPW* ("Nationaler politischer Wandel in entgrenzten Räumen") project, which examines changes in the structure of the national political spaces of six Western European countries under the current process of globalization (Kriesi et al., 2006).

Previously, entities and core sentences were annotated manually, by reading the entire article. COSA is devised to support this process by performing "semi-automatic" annotation. It should reveal the automatically recognized entities and suggests core sentences, but the final decision remains with the person who reviews the processed articles. COSA's mission can be likened to applications of relation mining (see e.g. Rinaldi et al. (2006)) and consists of the following three tasks:

1. Preprocessing newspaper articles
2. Recognizing predefined political entities
3. Extracting entity relations

The COSA project has been active for a while and all three tasks were previously implemented in a pilot study (Wüest, 2006; Wüest, Bünzli, and Frey, Un-

[1] The project was previously called CAN.

published). Nevertheless, a complete reimplementation was part of the present study, for the following reasons:

1. The previous implementation of the preprocessing module targets German newspaper articles, while the solution developed during this study provides tagging and parsing for English texts. It thereby extends the range of data that can be processed by COSA.

2. The remaining two tasks were not implemented in a language-agnostic way, making it impossible to use them for an English data set without modification.

3. The previous system is not, overall, designed to be very user-friendly and not very transparent in its handling, formatting and storing of data.

Even though none of the previous code was reused, it remains clear that this study is not about reinventing COSA, but merely about implementing predefined tasks while complying with predefined guidelines. The following specifications were not open to negotiation for the purpose of this implementation:

1. The format of the XML files supplied by the Department of Comparative Politics, which each contain a newspaper article and serve as the input to the system,

2. the format of the XML files that contain the annotated articles,

3. the syntax and semantics of the definitions of relevant political entities supplied by the Department of Comparative Politics, using either names or keywords,

4. and the reliance on open-source software.

While many linguistic decisions had to be made and theoretical reasoning will be provided in the course of this thesis, developing of a functional implementation for productive use by people without a background in software engineering or computational linguistics was much at the heart of this study. The following goals can thus be formulated:

1. The preprocessing module should parse the supplied articles and use a combination of third-party tools to annotate the article's text with correct linguistic information (part of speech tags, syntactic structure).

2. The entity recognition module should correctly interpret all supplied entity definitions and mark all entities that occur in the article with high accuracy.

3. The core relation extraction module should combine the entities into plausible core sentences and provide output in accordance with the specified format.

4. The implementation as a whole should be easy to install and use, language-agnostic where possible, fit for productive use as well as transparent and open for future extensions.

In order to develop a solution, an English test corpus was obviously needed. It comes in the form of more than 700 articles from two British newspapers, *The Times* and *The Sun*. They will be referred to as the *uk_2005* set of articles, since they were all published in the two months prior to the United Kingdom general election of 2005.

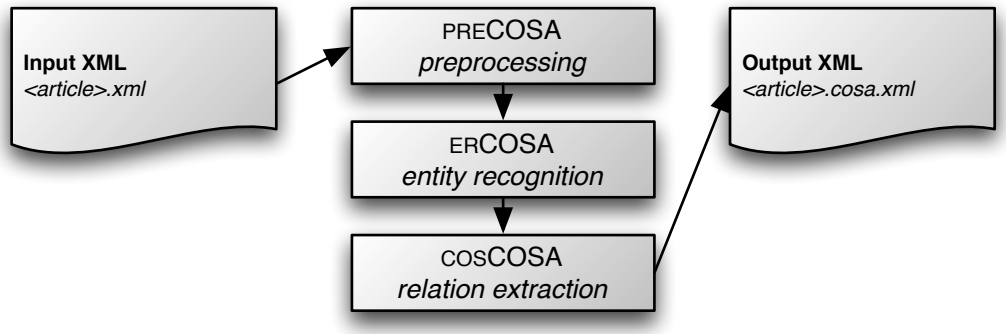

Figure 1.1: COSA implementation overview

Rather than spreading the effort evenly among the implementation of all three COSA modules, the emphasis was on the first (preCOSA) and second (erCOSA) rather than the third (cosCOSA). preCOSA, which is described in chapter 2, is important because it adds the capability of processing English articles to COSA. Linguistic aspects came into play in choosing the third-party tools and judging the effect of differences among them. It was also a considerable chore technically. The second module was the greater challenge, however.

Chapter 3 explains how erCOSA uses finite-state devices for a custom entity recognition scheme. Because erCOSA is the most substantial contribution of this study, it receives the most attention in this thesis. Lastly, cosCOSA is a very basic implementation. High-quality relation extraction is difficult to do and there was not enough time in the context of this study. Chapter 4 provides more details. Figure 1.1 gives an overview of the outlined COSA implementation. Following the discussion of the individual tasks, some concluding remarks will be made. The appendix contains the README and INSTALL as well as exemplary configuration files for the individual modules.

2 preCOSA: Preprocessing English newspaper articles

Preprocessing is an indispensable term to describe serial aspects of a system. This comes at the cost of a limited expressiveness and precision, however. According to the *Oxford English Dictionary*, to preprocess something means no more than "[t]o subject [it] to a preliminary processing". The question becomes obvious, then: preliminary in regard to what other process? We shall see that, in dealing with the sequence of steps that this section is about, there will be a lot of preprocessing on subordinate levels. Moreover, it is questionable to present our source material as "raw data". Newspaper articles clearly do not (yet) usually manifest as readily useable XML files. There has obviously been preprocessing outside of the scope we are here concerned with, which is negligible for our purposes. The preprocessing we speak of, then, is to be understood with regard to COSA's main tasks, i.e. the recognition of political entities and their relations (see chapter 3 and chapter 4). In the context of this study, preCOSA is the module which prepares the articles for erCOSA and cosCOSA.

2.1 Goals and choices

The goal of preprocessing data must be to prepare it for the next step along the way, as we have just established. More precisely, we should ideally do everything which in any way improves the performance of the task we are preparing for, while not doing anything which is of no use and a waste of effort. In reality, one usually cannot do all that would benefit the outcome in some areas while doing more than is strictly necessary in others. We shall first describe possible preprocessing tasks, so we can later establish whether such is the case with preCOSA.

If we think about the requirements for recognizing political actors and topics, they are not so different from what applies to many "higher-level" NLP applications. There is in fact a set of fairly common "shallow" text processing operations

which can all be seen to aim for text normalization of various types (Neumann and Piskorski, 2002):

Sentence splitting While our input data tells us where new paragraphs begin, it does not contain such markers for sentence beginnings. As we shall see (subsection 3.1.2), COSA's definition of entities alone requires the demarcation of individual sentences.[2] It is also a prerequisite of the following tasks.

Tokenization Since COSA defines the relevant entities on the level of individual words, our sentences need to be broken down into suitable units. Again, this is also a prerequisite for tagging and parsing.

Lemmatization is another process which our efforts depend on. While it is not very relevant at all for person names, the reduction of words to a normalized base from is paramount to enable us to match keywords without having to worry about their specific appearance in the text (e.g. "child" vs. "children").[3]

POS Tagging is somewhat more complex than the previous processes. Furthermore, taggers often lemmatize their input at the same time. While it is conceivable to manage without this information, we should at least exploit the tagger's ability to assign a distinct part of speech tag to proper nouns, as ERCOSA indeed does (see section 3.7).

Parsing or syntactic parsing, to be more explicit, tries to capture the structure of a sentence in terms of the language's grammar (Carstensen et al., 2004). While the current COSA specification does not allow this (see subsection 3.1.2, it might certainly be interesting to be able, to define a topic by the co-occurrence of two keywords in the same noun phrase, for example. More importantly, an intelligent implementation of the relation extraction module (see chapter 4) definitely requires syntactic information. Note that

[2]Notably those that are specified by keywords which need to co-occur in the same sentence.

[3]The reason I except person names is that they are not subject to inflection, except for possessive forms ("Gordon Brown *'s* policy"), which are usually isolated during tokenization. While this is true for English, it does not hold for many Slavic languages like Russian, Czech and Polish.

chunking and *head extraction* are often seen as separate tasks which parsing proper depends on.

Named Entity Recognition This might seem somewhat out of place, since our main task of recognizing political actors and topics can be seen as an instance of entity recognition. Ours is far from an all-purpose kind of ER, though. We only seek to recognize a set of predefined entities, which do not necessarily conform to what is usually called "named entities" (see section 3.1 for details). It could be useful to use a tool which is broader in its scope to normalize locations, brands, companies and the like. Still, with the whole process in mind, this is probably of lesser importance.

Coreference Resolution would be a great help, on the other hand. The aim, in general terms, is to map linguistic objects to a template instance if the former can be said to refer to latter. According to Carstensen et al. (2004), subtypes include the resolution of proper name references ("*Gordon Brown* speaks out. *Brown* says ..."), pronominal references ("*He* says ..."), and designator references ("*The prime minister* says ..."). All of these are of great value for an implementation of COSA. Unfortunately, this remains a big challenge for the time being.[4] Hence, in contrast to most of the previously mentioned tasks, coreference resolution is not considered a standard procedure for which a host of good quality tools are available.

Looking at this list of potentially rewarding preprocessing tasks, some of the choices and challenges that any given implementation must face become apparent. More often than not, there are a number of valid alternatives when choosing a tool for one of the tasks. These may not just be different implementations of the same approach to the problem but use differing rationales which in turn have their respective advantages and disadvantages. The biggest challenge, however, probably consists of stringing together all these processes and their respective tools. The individual tasks not only follow each other but are also highly interdependent. At every junction, there is the potential for difficulties regarding the transition from the output of one process to the input of the next without

[4]Especially the resolution of non-pronominal references proves difficult. It requires knowledge on various levels, from morphology up to semantics and pragmatics (Mitkov, 2003).

losing any useful data.[5] Added to this, we might want to carry non-linguistic data through the entire process and should end up with consistently and usefully formatted output. If the processing pipeline that results from conjoining the various sub-tasks involves tools which are complex and particular about how they except their input, a smooth setup is a major chore. Such are the challenges that face preCOSA.

2.2 Pipeline design

2.2.1 preCOSA and NLPPL

The first part of the implementation that was developed in this thesis, preprocessing the newspaper articles to prepare them for the actual annotation, is the only inherently language-dependent COSA module. A new implementation was therefore inevitable if COSA was to be expanded to English data. While the aim is to build a system that is comparable to the existing German counterpart (see Introduction), it is also evident that most of the contributing tools will be different. On the one hand, the availability of free NLP tools for English is probably higher than for any other language. On the other hand, there are not many full (deep) linguistic parsers that are fit to handle "wild" texts like newspaper articles that use long sentences, varied style and a broad lexicon. Fortunately, Pro3Gres, a robust, fast, deep-linguistic dependency parser, has been developed by Gerold Schneider at the University of Zurich (Schneider, 2004). This parser expects its input data to have undergone most of the preliminary preprocessing tasks mentioned in the previous section. It focuses on finding dependencies between the heads of noun and verb group chunks. Pro3Gres has already been used to analyse large amounts of biomedical research documents in order to enable the mining of relations between proteins. For this specific application, and to be able, at the same time, to use the parser with a variety of different input documents and a choice of tools, it has become the centrepiece of an integrated natural language processing pipeline (NLPPL), developed by Kaarel Kaljurand (Rinaldi et al., 2007).[6] Under the circumstances, and even though syntactic

[5] Not every step might in reality depend on the result of the preceding one; it might also be based on an earlier output.

[6] It has not become entirely clear to me whether NLPPL is the actual name of the implementation, or whether it is to be thought of as one implementation of just that kind pipeline,

parsing is not, at least at the current stage, essential for COSA, NLPPL was an obvious choice for the preprocessing framework of this project. Figure 2.1 is a simplified sketch of the result, which goes by the name preCOSA. We shall now briefly look at the structure of NLPPL and the issue of integrating third-party tools and will then discuss each of the sequential steps in more detail.

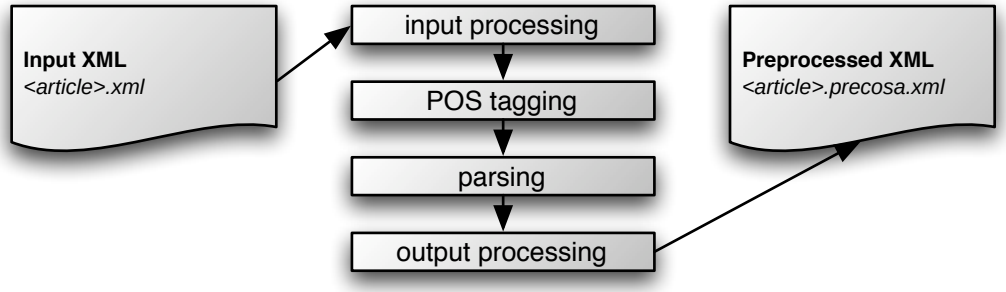

Figure 2.1: preCOSA Overview

2.2.2 Third-party tools

Using NLPPL as the basis for preCOSA, greatly simplifies the task at hand. All the implements for building a custom pipeline are already laid out and ready to use. The parser (Pro3Gres) in particular is well-embedded and feeding it appropriately formatted input as well as handling its output is taken care of. It still was a long way from NLPPL to erCOSA: some tools were substituted for others, scripts had to be extended and the COSA-specific input and output requirements had to be met. Thankfully, the flexibility of NLPPL allowed all this without pushing its limits. At its core, NLPPL consists of a XML build file for Apache Ant, a Java-based build tool working with dependencies.[7] In the end, Ant executes a sequence of commands (scripts, applications, etc.) necessary to build a specified target. NLPPL defines some targets (e.g. *postagger, parser*) as well as the format of the XML files which represent documents that have been processed up to the corresponding point in the pipeline. This allows extending

even if the abbreviation is not an established one. I will, in any case, use it in the former sense when referring to Kaarel's work, while preCOSA is the name given to the result of customizing it for COSA.

[7] `http://ant.apache.org/`

the pipeline with reasonable effort and have it deal with custom input, use custom tools and deliver custom output.

There are a few general aspects worth considering when we speak about the integration of external tools into our pipeline. We said earlier that there often is more than one tool for a particular task, so we are left with a choice to make. One obvious factor in favour of a particular tool is a pre-existing integration with NLPPL. But there are more, and arguably more important, criteria:

1. Good quality results
2. Ability to handle our input and produce a useful output format
3. Readily distributable, few dependencies
4. Non-restrictive licence

The first point sounds straight-forward, but as so often, it is the evaluation which is difficult. This is especially true in this case because the performance of the sentence splitter influences that of the tokenizer, which is the basis for the tagger, which in turn (and to an even greater extent) is what the parser depends on for its own work. It was beyond the possibilities of this study to systematically compare the quality of alternative tools. The focus was therefore on the second point (input/output). While a lot can be done by pre- and postprocessing the data we feed to and retrieve from an application, we shall see that some differences can be crucial. The third point is there for practical reasons, as one of the stated aims of this implementation was to be (more or less) easily deployable on other Linux systems. The last point, licensing, is of no technical or linguistic relevance, but COSA requires all software to be free, at least for academic use.[8] We will return to these criteria in the following discussion of the individual tools.

2.3 Input processing

As we go through the sequence of steps which our data takes as it passes through the pipeline (see Figure 2.1), we need to look at our input data first. It is necessary to make it conform to a format which the pipeline understands.[9] Consequently, our first goal is to reach the target *inputfilter-cosa*, which should yield tokenized

[8] "Free" as in "free speech", that is, as opposed to "free beer", to use the famous analogy. Having said that, the reality is that the software involved should rather belong to both categories...

[9] "Understand" is probably saying too much, in fact, since the data format of intermediate results are in no way set in stone by NLPPL. Remember that it is, at its core, no more

text in the expected XML format. Figure 2.2 gives an overview of what is involved in this first step of preprocessing. The starting point for our work are

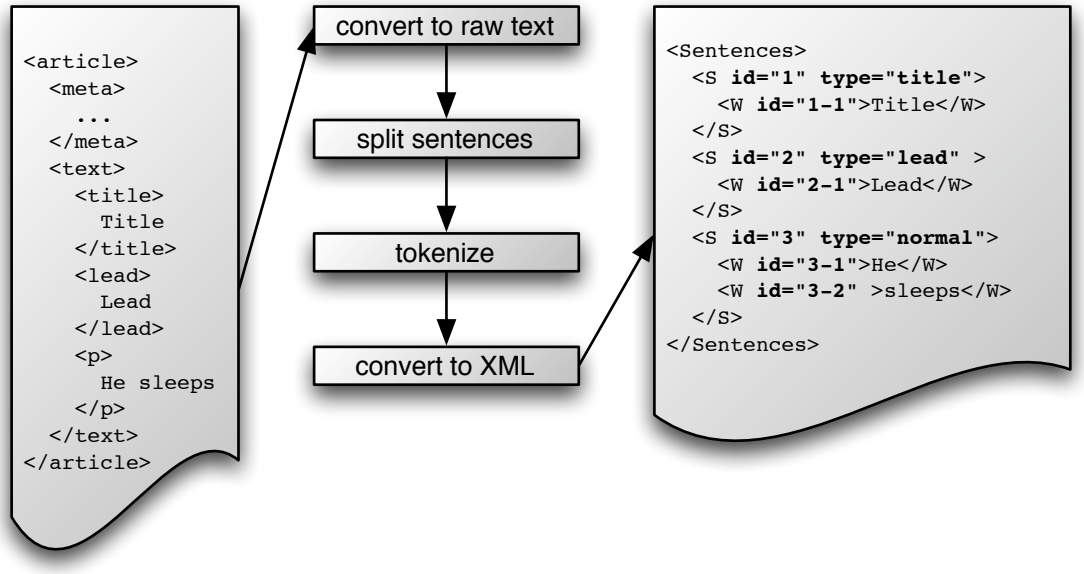

Figure 2.2: preCOSA Input Processing

newspaper articles in a COSA-specified XML format. The basic structure of these documents will still be the same for the end result, after all processing has been completed. The root element *article* contains the two subtrees *meta* and *text*. As one might expect, the former contains meta-information about the article, like its newspaper, its date of publication, its author, and so on. We will leave this section untouched. The *text* section contains all the information that is of interest to us, namely the article's text itself. Paragraphs are separated and marked as either a title, a lead[10], or a regular paragraph. This information needs to be preserved and still be part of preCOSA's output. In the first part of preprocessing, which we will call *input processing*, we will tokenize the article's text, express the result in NLPPL-conform XML and make sure we do not lose information about paragraph boundaries and types in the process.

XML-formatted data may be increasingly popular in NLP, but this is true

than instructions for an Ant build process. I should probably say "accustomed to" instead. The point is that it does make a lot of sense to use the prevalent format because that allows reusing most of the existing scripts which are the flesh around the pipeline's bones, really.

[10]"A summary or outline of a newspaper story", according to the *Oxford English Dictionary*.

mainly for higher-level applications (like erCOSA or indeed COSA as a whole). Basic tools like sentence splitters and tokenizers usually work best on "raw text". That is why, even if the result of input processing will again be an XML file, we first extract the relevant textual information from the input document. The solution to not losing track of where a new paragraph (of one of the types introduced above) starts, is to insert textual markers in the corresponding positions (e.g. `#title#`, `#lead#` and `#p#`). These will have to receive special treatment once our data is converted back into XML.

Sentence splitting and tokenization are both handled by third-party tools, so we will have to apply the criteria which were laid out in subsection 2.2.2. The sentence splitting solution that is already integrated with NLPPL is Adwait Ratnaparkhi's MXTERMINATOR (Reynar and Ratnaparkhi, 1997), a tool written in Java and part of a suite primarily meant for POS tagging. Unfortunately, this solution posed some problems with regard to the last two acceptability criteria, distribution and licensing. It basically consists of a Java class with somewhat intransparent dependencies. It also features a non-standard, slightly prohibitive licence.[11] A replacement which does not suffer these drawbacks and seems to perform just fine comes in the shape of sentenceBoundary.pl, a Perl script by Marcia Muñoz.[12] In the case of word tokenization, NLPPL's "stock" tool, a Sed script (as in *s*tream *ed*itor, the well known POSIX tool) by Robert MacIntyre, is a viable choice.[13]

We now have the output of the tokenizer, a file with one sentence on each line, space-separated tokens and paragraph markers. We should transform all this into an XML file that meets both the NLPPL "standard" format as well as some COSA-specific requirements. As was pointed out already, NLPPL does not formally define a compulsory format for the intermediary milestone XMLs, but scripts are in place to deal with documents of a certain structure. This structure requires a root element (*Sentences*), which contain sentence elements (*S*), which in turn contain token elements (*W*). Sentences and tokens should have

[11] The licence grants the right for educational and research use only to the individual who downloads the software. Any kind of redistribution or the use by people who are not under direct supervision of the downloading person is not allowed.

[12] `http://l2r.cs.uiuc.edu/~cogcomp/atool.php?tkey=SS`

[13] From the source of the script:

```
# Sed script to produce Penn Treebank tokenization on arbitrary raw text.
# Yeah, sure.
```

unique IDs. COSA requires sentence and token IDs to be of a specific format and paragraphs to indicate their type.[14] An according XML file (see Figure 2.2), which is the end result of our input processing and meets all these requirements, is created by a Perl script which had to be extended to handle the COSA specifics.

2.4 POS tagging (TreeTagger)

With preliminary input processing out of the way, we can now feed the tokenized text to further tools, which assign a part of speech and a lemma to each token. The stand-alone solution for tagging that was used with NLPPL before is MXPOST (Ratnaparkhi, 1996), a close relative of the sentence splitter mentioned in the previous section. Again, there are the same issues with its licensing and ease of distribution. The lemmatization of tokens tagged by MXPOST was previously carried out by MORPHA (Minnen, Carroll, and Pearce, 2001), a tool for morphological processing. The problem encountered here is to do with the second of our acceptability criteria for external tools (see subsection 2.2.2), the shape of the output. It seems that MORPHA invariably returns a lowercase lemma. If we want the options of exclusively using the normalized text of a given article for further processing, then this is detrimental to the results. Clearly, useful information would be discarded. Consider this exemplary sentence:

(1) *His ally was, again,* Black.

Even if we lost the capital letter of the name, we might of course rely on the tagger to get the part of speech right, which would still allow us to distinguish between a person with the surname "Black" and the colour "black". But firstly, taggers aren't infallible, especially with proper names, and secondly, missing one of the political actors altogether, as a consequence, is a pretty serious flaw.[15] It is preferable to not have to rely on correct POS tagging and be able to match names and keywords with respect to their capitalization. In summary, an alternative to the MXPOST/MORPHA combination was needed.

[14] The ID has to be of the form *article_identifier-sentence_nr-token_nr*, e.g. "thetimes_2005_05_02_449-5-13". Note that this was abbreviated in Figure 2.2. The paragraph types were introduced previously in this section.

[15] For the recognition of entities, we must achieve a high recall figure.

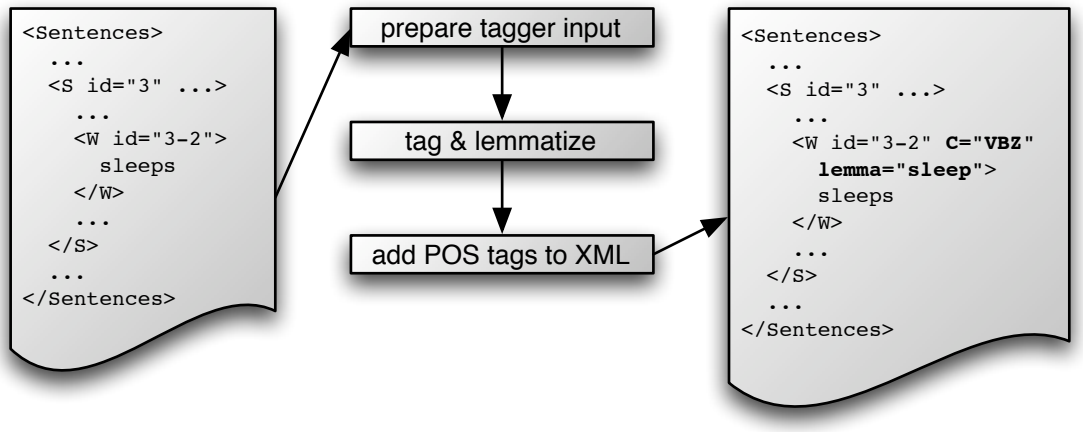

Figure 2.3: preCOSA POS Tagging

A substitute was found in TreeTagger (Schmid, 1994), which does both POS tagging and lemmatization at the same time. It uses a probabilistic approach to tagging with the help of a decision tree and is available for other languages, among them German, as well.[16] Even though it retains a word's capitalization for the lemma, there is yet another capitalization issue for which a solution had to be found. It is customary, at least with English print media, to use all capitals for the first few words of an article and each of its paragraphs. This does present us with a problem in matching names or other capitalized words which might be part of one of the political entities we will later be looking for. Assume we want to find Gordon Brown and have the following sentence in our article:

(2) *BROWN HAS ALLUDED to the possibility of reconciliation.*

Searching for the surname "Brown" will obviously not succeed, and unfortunately, lemmatization will not help either. If TreeTagger cannot find an all-caps word in its lexicon, it will try again with a lowercase version. Consequently, it will find the word and give us the lemma "brown", but even if that word was not a known adjective, we would still get the (unchanged) lowercase version as the lemma. This will not match our search term either, and we are back to the situation we had with MXPOST. Another approach would be to generally do case-insensitive matching, but "brown" again exemplifies the problem. It is not confined to person names, either: "(a) conservative party" is by no means the

[16] In fact, it is partly used for preprocessing German articles within the COSA project.

same as "(the) Conservative Party". We have already established that we do not want to rely on part of speech tags to find our entities, and also that we must not miss them.[17] The solution is to alter the input tokens for the tagger. More precisely, we use regular expressions to capitalize successive all-caps words at the beginning of a sentence:

(3) *Brown Has Alluded to the possibility of reconciliation.*

While this is not perfect, it seems to produce the fewest tagging and lemmatization errors and makes sure we will not overlook any entities.[18] With this modification in place, TreeTagger now outputs useful tab-separated values from which we can gather the POS tags and lemmata.

The incorporation of TreeTagger's results into the next milestone XML (target: *postagger*) is handled by a Perl script which was originally written for MXPOST and had to be adapted. One modification is notable in particular, because it shows the interdependence of the tools that make up our pipeline. TreeTagger uses a refined version of the Penn Treebank tagset to distinguish between the verbs *to be*, *to have*, and all others.[19] Our parser, Pro3Gres, uses the same tagset but is unaware of the refinements. We therefore have to translate all non-standard tags back into their canonical form. Having taken this final hurdle, our article now provides a lemma and a part of speech tag for each token. This information is needed by the parser, but it is also, as we shall see in detail later, of immediate use for recognizing political entities.

2.5 Parsing (Pro3Gres)

preCOSA has profited greatly from the basic framework and previous customization that NLPPL provides. The fact remains that the pipeline's *raison d'être* as well as the initial motivation to base preCOSA upon it, is its integration with Pro3Gres. Because the parser has not yet been trimmed for convenient usage or been released to a broader public, it is somewhat unwieldy in its handling. Thankfully, it is embedded in various scripts and tools which do all the

[17] Moreover, keywords that are not person names could be of any class.
[18] It should be noted that the article's text is not permanently changed. It is merely the input to TreeTagger that is modified.
[19] There are even some more, undocumented differences.

necessary pre- and postprocessing work. Thanks to the previous steps, we can now supply a properly tokenized, lemmatized and tagged text in the expected XML format. Consequently, no modifications were necessary to the whole parsing process. This is one reason why this step shall not be discussed in great detail. The other reason is more unfortunate. The syntactical structure we uncover by parsing the articles is not relevant to erCOSA, where we recognize the political entities, but should come into play when we look for relations between them in order to build core sentences. Because of different priorities and limited time and resources, cosCOSA is but a "catch-all" prototype implementation which does not presently make use of such information. Nevertheless, the syntactic dependencies greatly enrich preCOSA's output and are of potential use for many kinds of deeper analysis.

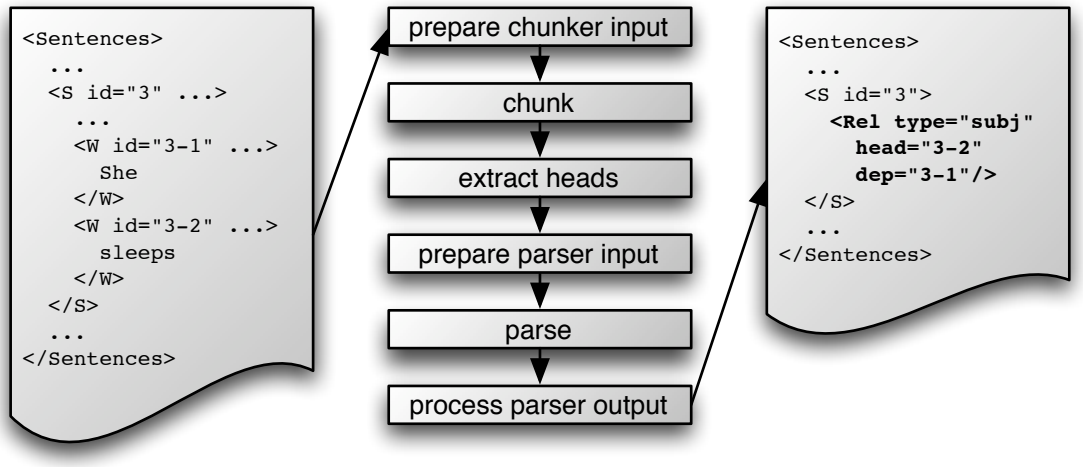

Figure 2.4: preCOSA Parsing

As you can see in Figure 2.4, Pro3Gres (preferably) only does the parsing proper (deep parsing) and therefore depends on additional information which we need to gain beforehand, through *chunking* (shallow parsing) and *head extraction.*[20] The goal here is to "stay as shallow as you can", which is to say that both chunking and, further up the pipeline, tagging, are low-level, non-recursive linguistic tasks that can, for example, make use of efficient finite-state methods. The results of these relatively fast processes significantly reduce the the workload for

[20]This "division of labour" is, of course, by design (Schneider, Rinaldi, and Dowdall, 2004).

PRO3GRES (Schneider, 2004). *Chunking* is the assignment of phrasal tags to the co-ordinate partial structures of a sentence. The smallest set of commonly used chunks are noun phrases, participial phrases and verb groups (Carstensen et al., 2004). In fact, the results of shallow parsing are sufficient for many purposes. Neumann and Piskorski (2002) suggest that "the shallow parser [be] used as an efficient preprocessor for dividing sentences into syntactically valid smaller units, where the deep parser's task would be to identify the exact constituent structure only on demand." PRO3GRES, as it is embedded in NLPPL, bases its parsing on LTCHUNK's (Mikheev and Finch, 1997) annotation of noun and verb groups.[21] Since PRO3GRES parses only between the heads of chunks, a relatively straightforward pattern-matching script written by Gerold Schneider is used to extract those heads. Finally, the parser's output, which consists of Prolog clauses that describe the syntactic dependency structure of the parsed sentences, is merged into the XML file which is the result of this step (target: *parser*). Our data must be transformed before and after chunking as well, but all the necessary code was already in place and the particulars are not of interest for PRECOSA.

One of the goals of basing PRECOSA on NLPPL was to enable the use of PRO3GRES, a fast and robust deep linguistic parser, with little hassle. In succeeding to do so, we can now claim to offer rich linguistic information that could potentially be used for many higher-level applications. Admittedly, the quality of the parsing was not investigated because it turned out not to be immediately relevant for the current implementation of ERCOSA or COSCOSA. PRO3GRES has, however, previously shown its good performance in the difficult domain of biomedical research literature (Schneider et al., 2004). Our newspaper articles are much more similar to the largest share of texts in the Penn Treebank, which is the primary statistical backdrop of PRO3GRES. We can therefore safely assume good results overall.

2.6 Output processing

The *parser* target of PRECOSA, which is the final target involving actual linguistic processing, yields an "NLPPL-style" XML file that describes the syntactic

[21] LTCHUNK, unfortunately, is one of the major stumbling blocks when trying to deploy NLPPL (and thus PRECOSA) on a new system. It is available in binary form only and no longer maintained, which means it does not run on current systems without tinkering.

dependencies found. But it does not contain everything we need to produce a "COSA-style" XML document, which holds the article's meta information (newspaper, publication date, etc.) as well as the article's text and all the linguistic information we have discovered (sentence boundaries, token boundaries, lemmata, POS tags and syntactic structure). There is also the question of how to represent the results, since we want preCOSA's output data to be consistent with COSA specifications. This benefits modularity as well as manageability.[22] The last step of our pipeline (target: *outputfilter-cosa*) is the combination of two tasks, then: combine data which is distributed among several files, and store it in an output XML file of the required format.

The article's unchanged meta information is contained in the XML file that was the original input to preCOSA, the tokenized and tagged text is in the XML file produced by the *input processing* and *POS tagging* steps, and the syntactic annotation can be found in the the file that resulted from the *parsing* step. We therefore need to merge these three XML documents into one (see Figure 2.5). What makes this (slightly) less trivial is the use of different element and attribute names and a different hierarchical structure in the resulting file. While no XML schema has been formally defined for the output of the preprocessing module, it makes sense to adhere to the COSA specification where possible, because we will ultimately have to comply with it anyway. Said specification defines a *type* attribute for each token, which can assume the values *normal* or *punct*. It is the only required information that is not explicitly part of the data at hand. Luckily, we can infer this "type" from the POS tag of the token in question. Along with the terminology changes and the correct merging into the resulting document, this is handled by an XML transformation using XSLT. We have thus reached the end of our pipeline. The newspaper articles are output as clearly structured XML documents containing rich linguistic annotation.

[22] Remember that preCOSA is language-specific, so it is the module most likely to have several implementations. For the sake interoperability, a common output format is of paramount importance.

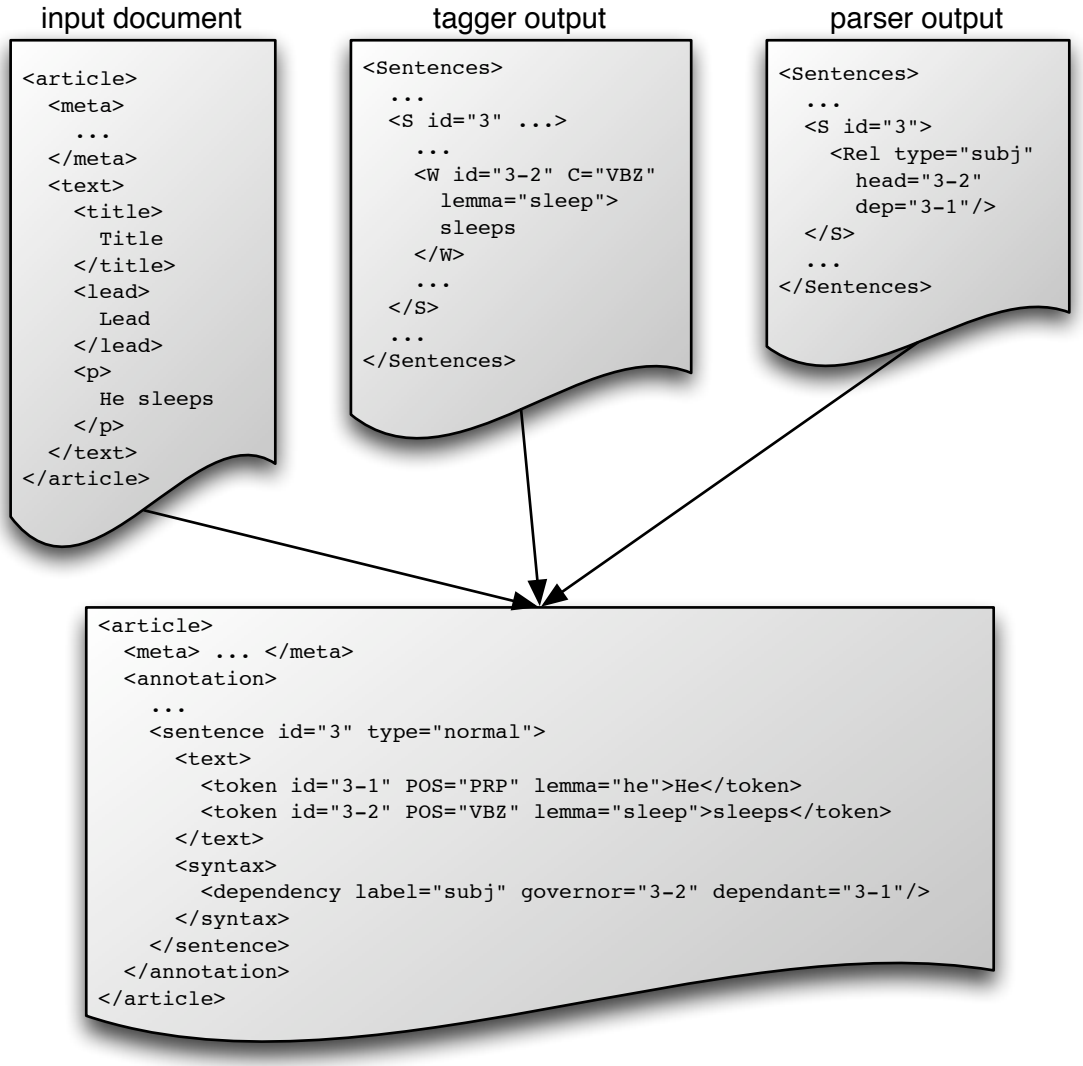

Figure 2.5: preCOSA Output Processing

2.7 Assessment

When we try to assess preCOSA, and this will hold for the other modules of this implementation as well, the idea is not one of a strictly empirical analysis that relies on quantitative data. The figures resulting from such investigations would certainly carry some interest, like the share of incorrectly tagged tokens, the number of lemmatization mistakes or the correctness of the syntactic parsing. The reason for not presenting this kind of evidence is twofold. On the one hand, we lack any kind of "gold standard" for our articles to check against.[23] On the other hand, it must be said that preCOSA should be viewed as a collaboration of various pre-existing tools, which have proven their individual performance before. What deserves our attention, then, is not the quality of those third-party applications, but an evaluation of how well preCOSA, as a pipeline that consists of a selection of tools and a number of scripts that manage their interoperability and data representation, prepares the original newspaper articles for COSA's higher-level tasks.

As for the selection of tools for our pipeline, there are two aspects to it. Firstly, we have decided to include some NLP tasks (e.g. parsing) and exclude others (e.g. coreference resolution). Secondly, we have chosen particular implementations of those tasks over others (e.g. TreeTagger over MXPOST). Looking at the tasks handled by preCOSA, there are those which are indispensable, namely sentence splitting and tokenization. Clearly, the sentence and token units are the basis for any kind of further processing. Lemmatization as the means to normalize word forms is a requirement for erCOSA too. The case for POS tagging is a bit weaker. One can imagine doing the kind of entity recognition that we need here (annotating political actors and topics) without that information, and in fact, it was used only minimally in erCOSA (see section 3.7). However, since a non-trivial implementation of the relation extraction module would greatly depend on syntactic analysis, which in turn relies on POS tags, it is more than justified. This leads us to syntactic parsing. We have just established that it is the stepping stone for generating high-quality core sentences, but we have also pointed out that cosCOSA does not presently make use of the parser's results. Going back to our list of potentially rewarding linguistic tasks (see section 2.1), this leaves named entity recognition and coreference resolution, which are not part of preCOSA.

[23] The effort of creating one is entirely beyond what is feasible.

The former might have helped in two ways. If it was used before the tagger, it could reduce tagging and therefore parsing errors. It might also take some work off erCOSA, since personal names, which are a large share of what erCOSA is dealing with, would already be marked as such.[24] On the whole, however, one should only expect modest advantages by doing named entity recognition in the preprocessing module. Coreference resolution is a different case altogether; even a modest approach, which only resolves pronominal anaphora, could hugely benefit erCOSA. Currently, even simple cases like this pass unnoticed:

(4) *Gordon Brown held a press conference yesterday.* He *addressed concerns about the government's NHS plans.*

erCOSA itself does a very limited kind of coreference resolution which only covers the use of people's last name instead of the full name (see subsection 3.7.1). Clearly, this is not enough, and improvements in this area probably have the greatest potential. The current implementation of preCOSA has not explored this potential because no suitable tools are readily available. Moreover, it is to be expected that coreference resolution would add another level of complexity and potential problems to the system.

Finally, guaranteeing the interoperability of the tools used for preCOSA and a consistent and useful representation of the linguistically annotated articles were the main focus. For this purpose, some applications were exchanged for others, as outlined in the previous section. While no quantitative evaluation was possible, there has been a considerable effort to discover and fix systematic irregularities like the case and tag issues described in section 2.4. There was also an emphasis on making sure that preCOSA is robust regarding its output. As long as the input articles adhere to the agreed format, there should always be an equally well-formed, COSA-conform and formally correct output.[25] It is such qualities that, besides the linguistic performance, are just as important (if not more) for the productive use of a system. Overall, the lack of coreference resolution seems to be the biggest and only decisive shortcoming of preCOSA.

[24] Ideally, this would hold for political parties and organizations too, but one would have to run actual tests to see how well all that works.

[25] More than 700 unique articles were processed by preCOSA during its testing.

3 erCOSA: Recognizing predefined political entities

Recognizing the political actors and topics that are relevant to studies within the *NPW* project for a given collection of newspaper articles is the pivotal task for this implementation of COSA, both on the technical and the linguistic level. In a stark contrast to preCOSA, the preprocessing module which was described in the previous chapter, the resulting program here is not based on a previous implementation or a proven concept to tackle the challenge at hand. It uses an original, maybe even somewhat unorthodox method that had to be developed from the ground up to cope with the requirements that were set forth in terms of linguistic capability and flexibility as well as reliability and robustness when processing large amounts of data. Since most of the time spent on this project was dedicated to the implementation of this task (erCOSA), which, referring back to Figure 1.1, is the middle module of the overall system, we shall have a detailed discussion of of the task itself as well as the solution that erCOSA offers.

3.1 Political entities

First of all, we will describe what political actors and topics are, how they are defined for the purpose of COSA and how we can describe the problem of recognizing them in our preprocessed newspaper articles.

We will call political actors and topics *political entities*, or in short, entities. This leads to erCOSA being referred to as a process of entity recognition. It was mentioned before that we must differentiate between this task and what is commonly understood when one speaks of (named) entity recognition (see section 2.1). The political actors we are looking for would fall under the category of named entities, because they are either people or political bodies (mostly parties), but that is not true for political topics, which are much more varied.

More importantly, while the more common task is to figure out how to generally discern named entities in a text and mark them, we get a list with all the political actors and topics that are of interest. We only need to find those, but we should find all of them and not make any mistakes. This is a very different task, of course, and the exact nature of the problems we are facing is largely determined by how the individual political entities are defined. In other words: how are we told what to look for? It is important to note that the formal aspects of defining actors and topics for this purpose were not open to much debate at this stage, but mostly specified at an earlier point in the overall COSA project. That it not to say that they are obviously flawed, but rather that they should be seen as a sensible make-do attempt that is, however, outside of the scope of optimization for this particular implementation. It seems probable that an evaluation of current results would show room for improvement.

The entity lists that are supplied by the Department of Comparative Politics are plain text files that define one entity on each line. Each record starts with an ID which uniquely identifies the entity. The ID is followed by one or more tab-separated *field=value* constructs. There are two different types of records, which have different fields: *name entities* and *keyword entities*.

3.1.1 Name entities

Name entities are used to describe people, i.e. political figures, which are of interest. The following are two examples as they appear in the actual entity list. The first is a straight-forward case while the second represents the complex end of the scale (tabs have been replaced by "‖" to improve readability):

(5) `act-6009-50063‖surname=Brown‖forename=Gordon`

(6) `act-6009-50031‖surname=Benn‖forename=Anthony‖forename=Tony ↩`
`‖surname=Wedgwood Benn‖forename=Anthony Neill`

As you can see in **(5)**, person entities are defined by the politician's first and last name. Unfortunately (for our purposes, that is), people tend to be referred to in a number of ways, as exemplified by **(6)**. There may be variations of a single name ("Tony" vs. "Anthony") and there may be multiple first and last names ("Anthony Neill", "Wedgwood Benn"). A variation of this are multipart names ("Marie Louise"), which, if separated by a space rather than a hyphen, pose the

same problem, namely one of tokenization. We would prefer to have to deal with at most two tokens for each person: the first and the last name. But since tokenizers usually rely on spaces for word boundaries, we must be prepared to deal with four tokens or more, which certainly does not make things any easier.[26] But this is not the only difficulty we shall return to when we discuss how erCOSA finds our politicians. What constitutes at least as much of a challenge is the fact that, more often than not, people in newspaper articles are referred to only by their family name, at least following their initial introduction. This not only increases the count of variations in which a given politician can appear in the text, but also introduces a great amount of ambiguity. While it is possible that we mistake a "Gordon Brown" for Britain's current Prime Minister, it is almost certain that we are making a mistake when we assume a random "Smith" to be one of the politicians on our list. Even if he is, though, we still don't know which of the four on that list is really meant. Because referring to people by only their last name is so common, we cannot afford to ignore these instances. erCOSA's solution to this problem, one of coreference, will be discussed later. For now, it will suffice to point out what textual occurrences we might want to map to the actor defined in **(6)** (please note that "‖" denotes a token boundary and we try to make as few assumptions about tokenization as possible):

1. *Anthony ‖ Benn*
2. *Tony ‖ Benn*
3. *Anthony Neill ‖ Benn*
4. *Anthony ‖ Neill ‖ Benn*
5. *Anthony ‖ Wedgwood Benn*
6. *Anthony ‖ Wedgwood ‖ Benn*
7. *Tony ‖ Wedgwood Benn*
8. *Tony ‖ Wedgwood ‖ Benn*
9. *Anthony Neill ‖ Wedgwood Benn*
10. *Anthony ‖ Neill ‖ Wedgwood Benn*

[26] To work around this, we could have fed all the names in our list to the tokenizer, or, if that was technically unfeasible, manipulate the text beforehand (e.g. by inserting hyphens as appropriate). There is, however, a major drawback. Such a solution would mean that the whole preprocessing operation depends on the exact content of the entity list. If, for example, a single politician were to be added to the list, all articles would need to pass through preCOSAagain. This was deemed unacceptable in the light of the availability of other solutions.

11. *Anthony Neill ‖ Wedgwood ‖ Benn*
12. *Anthony ‖ Neill ‖ Wedgwood ‖ Benn*
13. *Benn*
14. *Wedgwood Benn*
15. *Wedgwood ‖ Benn*

This is a pretty extreme example, but it is real and shows the potential for a complexity which cannot be ignored. Our implementation will thus have to infer all these variants from a given person record and have a way to assign the corresponding actor ID if one of them occurs in an article.

3.1.2 Keyword entities

Defining an entity by first and last name is certainly very fitting for finding people in an article, but when it comes to anything else, we need a more general approach: keywords. Names are different from keywords with regard to two important aspects: firstly, we can be certain that they always follow each other (in the sense that a given name directly precedes the family name and a middle name is surrounded by the other names) and secondly, we need to know whether we are dealing with a first or a last name, since, as far as politicians are concerned, only last names will usually occur on their own in newspaper articles. Keywords are all-purpose search terms without any inherent rules. It is therefore important to specify how they can be combined and what a given combination expresses, exactly. Incidentally, all entities which are defined by first and last name are political actors (as indicated by the ID prefix "act-"), but not all of the entities defined by keywords have to be political topics ("top-").

The examples below show records which define keyword entities: **(7)** is the trivial case of a topic with a single search term, **(8)** is an actor (a political party) defined by keywords and **(9)** lets us examine the full expressiveness of the allowed constructs.

(7) `top-5-5005034‖keyword=poverty`

(8) `act-6009-0‖keyword=Labour‖keyword=Labour Party`

(9)
```
top-5-5005049‖keyword=cot‖keyword=day nursery ↩
‖keyword=childcare‖keyword=child & care &!  medical
```

For **(7)**, the expected behaviour is fairly obvious. Whenever the token "poverty" occurs, we assign it the corresponding entity ID. **(8)** already raises further questions. Spaces are used for name entities as well, and we have intuitively taken them to mean that the separated words should occur in immediate sequence. The same holds true for keyword entities. In another parallel, we took multiple *surname* values to be alternatives, which is the same for multiple *keyword* constructs. It follows that an occurrence of "Labour Party" will always match both alternative definitions, but we will not want to recognize the underlying entity twice, so this is something erCOSA will have to address. Being allowed only to define topics in terms of a sequence of words is very restricting; we need more "operators" which allow for more flexible definitions. As can be seen in Table 3.1, which also indicates precedence, commutativity and associativity, there are currently three operators (including the "invisible" space operator).

	prec.	commut.	assoc.	description
$A\ B$	high	no	left	A immediately followed by B
A & B	low	yes	left	A and B in the same sentence
! B	middle	-	yes	anything but B (incl. nothing)

Table 3.1: Operators for keyword entity definition

The goal in specifying the syntax of valid keyword entity definitions is for the constructs to be as easy and intuitive to write and read as possible, while still allowing for considerable expressiveness. The rules for constructing a syntactically valid expression are the following:

1. There must be exactly one operator in between any two keywords, except for "!", which can only follow "&" and must not appear anywhere else.
2. A space only constitutes an operator if one is expected and there is no other operator.

This restrictive grammar has a number of advantages:

1. stray spaces do not impede correct interpretation
   ✓ `alpha beta  &gamma & ! foo  bar`
2. no association clashes, parentheses can be disallowed
   ✗ `alpha (foo & bar)`

3. no negation where we do not want it[27]

   ✗ `foo ! bar`

4. no purely negative definitions[28]

   ✗ `! alpha &! beta`

We should now be able to correctly interpret **(9)**. If we look at it as an instruction, it would translate into a sentence along these lines: "If a sentence contains 'cot', 'day nursery' or 'childcare', or indeed both 'child' and 'care' but not 'medical', then assign the ID 'top-5-5005049' to the relevant tokens." Unlike with names, we make no allowances for tokenization variations here; all keywords should be single tokens. They should also be normalized to match the lemmatized forms in our preprocessed articles.

It is possible that more expressive constructions could better define political topics. We might define an operator which means "near" and indicates a maximum distance between its operands or we might define relations which hold across sentence boundaries. From a linguistic perspective, it should be most promising to use syntactic criteria ("same noun phrase" for example), but the limit here could well be the acceptable complexity for the people who define the topics, rather than the technical and linguistic implications for our implementation, which we will look at next.

3.2 Implementation challenges

One might think that finding a number of politicians and topics in newspaper articles sounds like an easy job, given that we have a complete list of names and keywords to look for. At first, it sounds like something the *Find...* function of even the most basic text editor can handle. There are a few points which will lead us to reconsider this assessment, as discussed in the following list of functionality challenges:

1. *Entity definitions need interpretation*

   As can be seen from the list on page 26, the definition of an actor with a few

[27] The example here does actually make sense in meaning "*foo* followed by anything but *bar*". However, this functionality was not requested and might or might not prove useful for a future implementation.

[28] While this would be fairly easy to implement, it just does not make sense to assign a topic to a sentence by virtue of absent words only.

name variations can easily lead to many alternative textual appearances. We have to consider when looking for the politician in question. Clearly, there has to be an automated way of inferring all possible alternatives from the entity definition. With keyword entities, we even have to parse the keyword constructs to ensure that they are valid and finally, one way or another, translate them into whatever sort of data we need to match them in the article. In both cases, we need a process of interpretation, the specifics of which depend on how we go about recognizing the entities in the text.

2. *Finding is good, annotating is better*
   Building on the text editor example, suppose we have a way of entering just what we are looking for in the search box. Assume the entered tokens are found and highlighted. This is very well and a necessary first step, but that alone will not be of any help eventually. Recognizing an entity is to assign the respective entity ID to the tokens which represent that entity in the text. At the very least this means that we need to have a way of knowing which tokens were relevant for the match, identifying them, and remembering the resulting many-to-one relation.[29] This process could be described as *annotating* the tokens with the entity they represent. It also means that we might need to work with a more complex representation of the articles, one that contains not only their mere textual content, but meta information, too.

3. *Sentence and token boundaries matter*
   This is a comparatively trivial point, but it reinforces the necessity of linguistic meta information and hence a well thought-out representation of the article for our entity recognition scheme. We do not want to partially match tokens, and for complex keyword constructs (see the earlier discussion of keyword operators), we need to know whether the matching tokens are part of the same sentence.

4. *Redundant matches must be avoided*
   Since it is allowed to have multiple definitions of all possible entity record fields (*forename, surname, keyword*), we most likely run into multiple matches for the same entity on the same tokens. If we find a complete

[29] Remember that thanks to preCOSA, each token has been assigned a unique ID.

name, we do certainly not want an additional match for only the last name. It is the same with topics: if there is a specific multi-token match, we should ignore a secondary, less specific match.

5. *Coreference resolution for names*
   While not really a matter of principle, ambiguous last names are such a common problem for our task, that it is worth thinking about it in advance. Independent of the actual implementation, we will need a way of referring to earlier (or later) occurrences of the same name, in order to reduce the ambiguities. This means moving beyond the immediate scope of the current sentence, which will necessitate additional work of some kind.

6. *Complex keyword constructs*
   The use of the "&" and "!" operators introduces a new level of complexity. Entities so defined can spread over a whole sentence and the individual parts can appear in any order. Moreover, the definitions are no longer exclusively positive, in the sense that the non-occurrence of a token can be a binding condition. These provisions severely increase the complexity of the algorithm needed to match (or annotate) those entities.

It should be obvious by now, that our task is not as easy as one might think at first. We have to deal with challenges at many stages of the process: entity definition interpretation, article representation, matching algorithm, result collation and representation. In addition to meeting the expected capabilities, there are a number of technical goals for our end result:

1. *Extensibility*
   It might be necessary, in the future, to modify or add to the keyword operators and to have more linguistic information (e.g. syntactic structure) available during entity recognition. If we want such extensions to be easy to implement, we need a sound programmatic structure and transparent processes and algorithms.

2. *Flexibility*
   There are many parameters within the COSA project which are, for the time being, strictly specified but also subject to change, at some point. They include the structure and nomenclature of XML files, the format of

entity lists, the rules for article and entity IDs, and so on. A change in these parameters should not necessitate changing (many lines of) the source code. Similarly, erCOSA should at its core remain language-independent, with the language-specific parts clearly separated and readily extensible.

3. *Usability*
   While it is not necessary that the correct use of our implementation is obvious to anyone without previous instruction, users must not be expected to have in-depth technical knowledge and should be able to achieve their goal with just a few commands and interact with the program in a useful manner. Also, it should not be necessary to do time consuming operations unless strictly necessary.

4. *Stability*
   Users should not have to deal with crashes and where possible, conflicts should be resolved without user interaction, since typically, hundreds of articles will be batch processed. Helpful error messages and a log file should allow solving most problems.

These two lists are not meant to be exhaustive, but they give us a clear idea of the challenges erCOSA has to face, and the goals it should achieve. The remainder of this chapter describes the efforts to do just that.

3.3 Finite-state recognition devices

As the title suggests, erCOSA relies heavily on finite-state transducers (FSTs) for doing the crucial part of the work, namely matching and marking the individual entities in the articles. Finite state technology is widely used for natural language processing, but for the task at hand, there are no real precedents. Based on the criteria set out in the previous section, the final product might seem mildly unsurprising or even like an obvious solution to the problem, but previous approaches tell a different story. We shall first look at those attempts and the development towards the final result and then return to a more detailed discussion of finite-state devices and how erCOSA employs them.

For a start, we should note that there is a previous implementation of an entity recognition module for COSA, which was used for the first batch of Ger-

man newspaper articles: CAN_NER. On the positive side, it is implemented in Prolog and makes good use of unification algorithms by representing both entity definitions and articles as lists of tokens. The result is a fast and efficient system whose results can be tweaked by the addition of further "unification rules". The downsides are as follows:

1. *Outdated entity definition support*
   At the time this project started, the "!" operator was not yet part of the entity definition specification, hence CAN_NER at the time lacked support for parsing entity definitions which make use of it as well as the means to find those entities.

2. *Error-prone resolution of name ambiguities*
   CAN_NER implements a fast solution for resolving ambiguities by looking up previously asserted person entities and is quite inclusive in that it also expects first names to appear on their own. The drawback is that all names which occur on their own are mapped to all previously recognized actors with the same name. This leads to flawed results in a variety of situations.

3. *Limited flexibility*
   Unfortunately, CAN_NER incorporates language-specific rules in the main code and does not have the built-in facilities to easily switch between languages. It also relies on current COSA specification, e.g. the way article, sentence and token IDs are built. In places, it also suffers from Prolog's poor support for string manipulation and file I/O.

4. *Limited Usability*
   CAN_NER does not offer much in terms of user friendliness. There is no easy-to-use configuration, very little error handling and no logging. Makefiles keep all the parts together and handle batch processing. This comes at the cost of many intransparent interdependencies and a reliance on undocumented directory structures.

erCOSA does not guarantee a better solution to all these shortcomings, though it hopefully succeeded in most cases. In any case, the problems noted here were reason enough to build something new from scratch: erCOSA.

There were no restriction as to what tools, methods and programming languages should be used for implementing the entity recognition module, but since it was decided not base it on CAN_NER, Prolog made little sense and personal preference lead to a bias towards Python. Python provides convenient means to read and compose XML files, parse entity lists, manipulate strings and offer a convenient user interface as well as logging and error handling facilities. Still, the method to use for erCOSA's core task, the actual entity recognition, is no more obvious for it. What we need is a way of method of matching one bit of textual information (the definition of an entity) with another bit of textual information (the article's text).[30] As soon as the requirements for a match are more complex than the simple congruence of two single strings, this becomes a non-trivial task. Exploiting Prolog's unification formalism is one way to go, but not the only one.

As is often the case, the final version of erCOSA developed within this study was preceded by inferior attempts which ultimately lead to a better understanding of the problem at hand. Even the first try already used a finite-state approach, albeit in a much more limited way than the later incarnations. The use of finite-state devices is not an entirely surprising move. We are asked to recognize textual patterns, after all, and if these patterns are regular languages, then they can be represented by finite-state automata (FSA) (Roche and Schabes, 1997). Regular languages are commonly described with the help of POSIX regular expressions, and like most programming languages, Python does offer their use for pattern matching. It is possible that the underlying implementation of regular expressions does not rely on FSAs, but they are impractical for another reason: regular expressions are easy to handle but not very flexible in terms of combining them in a modular way. If we want to represent complex patterns and process not only words but also IDs, POS tags and so on, they become very cluttered and unwieldy. Even though the first attempt at using FSA-like devices failed, we shall discuss it here as it sheds some light on how the final solution came about.

The initial idea was to use FSA-like objects that are in some ways more restricted than "full-scale" FSAs, but also more powerful in others. These objects are based on a simple Python code snippet which offers a way to construct and "feed" FSAs as well as the option to use facilities which go beyond the finite-state formalism, namely memory (which essentially elevates the resulting device to the

[30] Even IDs and linguistic meta information are ultimately text, of course.

realm of push-down automata) and all sorts of side effects through customizing its Python methods.[31] This allows keeping the complexity of the FSA itself very low and, because everything is pure Python, have it process complex data objects instead. In other words, this approach relinquishes most of the power of the finite-state paradigm in favour of the power of a higher-level scripting language. As you can see below, an entity as defined by **(10)** is transformed into the two FSAs in Figure 3.1.

(10) `keyword=alpha & foo bar`

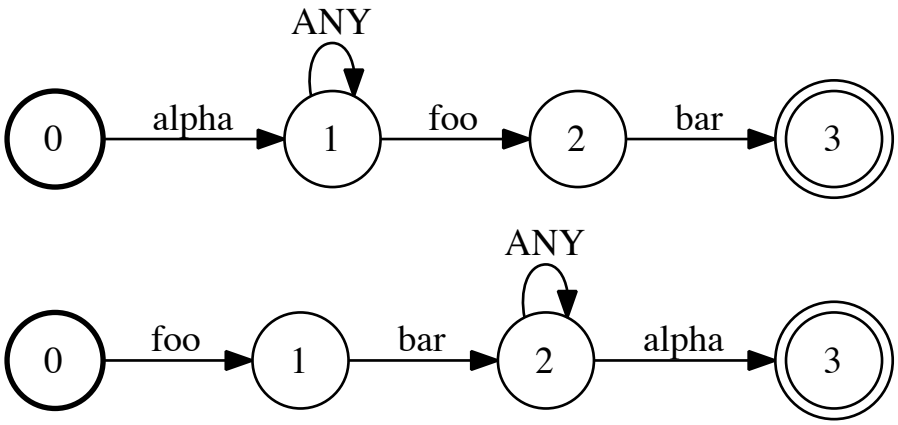

Figure 3.1: FSA test graph

The example shows the simplicity of the FSAs which are constructed. They are a very straight-forward transformation of the entity definitions, where the initial and the final state are connected by a sequence of transitions which each represent a token. The "&" operator corresponds to a cyclic edge which accepts any input token. Since the FSAs need to be constructed "by hand", i.e. each transition needs to be defined explicitly, the constructions need to be simple. The FSAs do not account for text before or after the relevant tokens, and apart from the "ANY" edges, there is only one transition from each state to the next. The free order of token (groups) that are joined by "&" is achieved by constructing multiple FSAs (see Figure 3.1). This simplified design allows for an open alphabet, so there is no need to enumerate all possible tokens, which would be impossible.

[31] `http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/146262`

This means that computing the complement of such an FSA is impossible, which also rules out intersections. This reduction of capabilities has to be compensated for. The code for FSA objects was extended so as to provide a feedback signal every time they process a token: either *FAIL*, if there is no transition for the next token from the current state, *CONTINUE*, if a transition was made but no final state was reached, or *SUCCESS* if the resulting state is in fact the final state. In essence, the only problem that is really solved within the finite-state realm, is the acceptance or rejection of a sequence of tokens with regard to a given entity. Everything else is left to be implemented with the more general means of Python. This includes feeding tokens to the relevant entity FSAs, interpreting their feedback, handling token IDs, managing sentence boundaries, assigning tokens to found entities, avoiding redundant matches and resolving coreferent names and other ambiguities. We shall see that with ERCOSA, all of this either does not apply or is handled within the finite-state formalism.

If the first implementation attempt can be said to use the finite-state paradigm only minimally, while ERCOSA does much more of the work with the help of finite-state devices, then why should the latter be preferable? In fact, that is not necessarily so. The former implementation did quite well, but ultimately failed in some regards (compare with the lists on page 29 and page 31):

1. *Inability to cope with complex keyword definitions*
   While a definition like the one in **(10)** is not much of a problem, it is not very elegant to have to "manually" account for the commutativity of the "&" operator, especially since this is well within the capability of a single FSA. Moreover, the restriction placed on the FSAs impede the implementation of the "!" operator, which was added to the specifications only later.

2. *Inability to efficiently resolve ambiguities*
   Because the entity FSAs, as they are used here, return all possible matches rather than a trimmed down set of relevant matches, the "surrounding" algorithms have to correctly reduce the results in the case of redundant matches and coreferent or irrelevant names. This requires a complicated flow of information and complex data objects.

3. *Poor transparency*
   As a direct result of the previous point, the whole recognition process uses

methods from different domains and is not very coherent and extensible at all.

It should be pointed out that this does not completely invalidate an approach which only uses finite-state machines to solve a very restricted subset of the entire problem. Some of the shortcomings are a result of the particular implementation that was chosen. In any case, the failure of this first attempt lead to the assumption that a much more inclusive use of finite-state devices might be worthwhile.

But what exactly does a "more inclusive use" of the finite-state formalism entail? Since finite-state devices operate on a sequence of symbols, all data we want them to process has to be "flat" information. We previously used complex objects, which incorporated the textual information of a token as well as "links" to the corresponding token, sentence or even article ID, the part of speech tag and eventually the political entity they represent. Now, we have to serialize all these aspects into a single text string (or a description of a finite-state automaton). There are two notable consequences to this:

1. *Need for transducers*
   In the first implementation attempt (see discussion above), we used FSAs as simple acceptors. Once a sequence of tokens matched or did not match, we used Python's extended facilities to provide according feedback and do further processing. In the approach taken by ERCOSA, we want handle an entire article at once and have finite-state devices do as much of the work as possible. This requires a way of marking tokens in the text, i.e. modifications to the input string. Finite-state automata are no good for this purpose; we need finite-state transducers, which represent a relation on strings (Roche and Schabes, 1997).

2. *Large and complex finite-state networks*
   Because we now have symbols (usually single characters) as the smallest unit, whereas we could work on entire tokens previously, we can no longer make do with a very modest number of states and transitions. Our FSTs need to account for token IDs, sentence boundaries, POS tags, entity IDs, and, moreover, incorporate rules to resolve ambiguities and eliminate irrelevant matches. Consequently, they are well beyond "manual" construction and, since they will easily contain thousands of states, no longer open to

insight by visualization.

In view of these points, it is apparent that we need a powerful finite state tool, one that handles transducers and provides a language to efficiently describe and combine them through various operations. Such a tool was found in SFST.

3.4 SFST and pySFST

Finite-state technology is frequently used in NLP, especially in the domain of morphological analysis and generation. Hence, a number of tools exist that cater for the respective requirements. Most widely used, and possibly most advanced among these is XFST, which is developed by the Xerox Corporation.[32] Unfortunately, it is a non-free commercial product and thus not suitable for COSA. A free (open source) alternative is available in the form of SFST by Helmut Schmid.[33] Even though this software, too, is aimed at morphological processing and includes tools which are only really useful in that context, it is at its heart an advanced finite-state processor and therefore fit for our purposes (Schmid, Fitschen, and Heid, 2004). SFST supports transducers, defines a programming language (SFST-PL) to define them and supports a wide range of finite-state operations (Schmid, 2005). Unfortunately, the SFST-PL syntax is quite different from that of XFST, so reading the respective transducer definitions is not necessarily intuitive. The relevant parts for erCOSA's operation will be explained as we go along.

Even though it is our stated goal to make extensive use of finite-state devices for our entity recognition, there are parts to the problem which require a different solution: reading and writing XML, transforming articles and entity definitions into SFST-PL, providing batch processing and logging facilities, and so on. Python is still the preferred choice for writing this auxiliary code. Luckily, Toni Arnold has developed Python bindings for the SFST library.[34] What this means, essentially, is that we can compile SFST-PL into a binary transducer from within Python, without having to resort to calling external binaries and using the file system as the only means of exchanging data between Python and SFST. It also allows accessing the transducers as Python objects that have their

[32] `http://www.cis.upenn.edu/~cis639/docs/xfst.html`
[33] `http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html`
[34] `https://gna.org/projects/pysfst`

own methods and properties. All in all, pySFST enables a more seamless and efficient use of finite-state devices within erCOSA.

3.5 Annotation by composition

So far, we have discussed methodological options, a previous implementation attempt, and the general approach finally taken by erCOSA as well as the tools to implement it. It is, however, as of yet unclear how our data (articles and entity definitions) are to be represented, and what kind of interaction will bring about the desired results. The former will be described in detail later on. For the moment, we will try to get a better understanding of the results we expect and how, systematically, we can attain them.

The idea is to have an FSA which represents the newspaper article, both the actual words and additional information like token identifiers, sentence boundaries and POS tags.[35] We then need a number of transducers (at least one for every entity definition) which annotate the article by marking the relevant token(s) when a given entity is found. The rest of the article remains unchanged. If no entity is "found" by the FST, it should output the unaltered article (and certainly not "nothing"). Such transducers can be built using the replace operator described by Karttunen (1995). Composing the initial article FSA with all entity FSTs in sequence, one after the other, results in a fully annotated article FSA from which the results can be extracted. While this method of using FSTs to annotate text is not well-established, Aït-Mokhtar and Chanod (1997) describe a somewhat similar application of finite-state devices to perform, in their case, shallow syntactic parsing. The challenge with this approach is to find a form of representing the text in an FSA that holds all relevant information while allowing efficient processing, and building entity FSTs which efficiently and correctly mark tokens which belong to the entity in question. The assignment of one or more tokens to an entity ID, finally, is what we call a recognized entity. These end up in the XML files that erCOSA outputs.

[35] We speak of an automaton rather than just a string because this allows for more flexibility. It is easily possible, for example, to allow for both the actual word form and the lemma, using a union ("mouse | mice").

3.6 Article FSAs

In order to be able to compose article FSAs with a number of entity FSTs, they all need to share a common, closed alphabet. Apart from this, we also need to agree on a common representation for tokens, sentences and annotations. Luckily, SFST has support for variables as well as includes (literal and binary). erCOSA defines the alphabet and global variable definitions in *common.fst*. The excerpts which follow are taken from the actual file. A few basics about SFST-PL syntax are useful at this point, especially where knowing the POSIX regular expression syntax is no help:

- Transducers can be assigned to a variable. Variable names are enclosed in dollar signs (`$Variable$`).
- New character classes can be defined; their names are enclosed in hashes (`#CharacterClass#`).
- Custom symbols can be defined; they are surrounded by angle brackets (`<symbol>`).
- The empty string is written as an empty symbol (`<>`).
- Whitespace is ignored in most places (unless escaped).
- A line break terminates expressions, unless it is preceded by a backslash.

3.6.1 Alphabet

(11)
```
ALPHABET  = [#Char#] [#Nr#] [#Delim#] [#Marker#] [#POS#]
```

Defining an alphabet for our finite-state devices is not as easy as it might seem. On the one hand, it needs to include all characters which could possibly appear in the article's text. Since COSA aims to be unicode-compatible, and SFST is able to process UTF-8 encoded strings, the characters which could potentially occur is vast. But every additional symbol in the alphabet increases the number of necessary transitions in the finite-state networks, so it makes sense to find a sensible set of characters which should cover most (English) texts. These are defined in a separate file and represented by the `#Char#` class. We can also define a shorthand notation (`$Text$`) for any word, i.e. one or more characters:

(12) `$Text$ = [#Char#]+`

On the other hand, we may want to define custom symbols which have special meaning. We could use "regular" characters for all our purposes, by specifying, for example, that the string "NEW_SENTENCE" marks sentence boundaries. Recognizing this string requires its own little FSA, however, while a custom symbol allows our finite-state devices to work much more efficiently and hence faster. All static content, then, should be made up of symbols as opposed to strings of characters. This is why erCOSA does not use token and sentence IDs as they appear in the input XML, since these are arbitrarily defined and not enumerable for practical purpose. Instead, we assign incremental sentence and token numbers to each token. These number symbols, e.g. `<132>`, are listed in a separate file (up to a sensible limit) and make up the character class `#Nr#`.

`#Delim#`, as the name suggests, contains delimiter symbols which mark, for example, where a sentence starts and ends. `#Marker#` contains only a single symbol at present and will be explained later on. `#POS#` stands for any tag of the used POS tagset which needs to be defined separately.[36] Each tag is a single symbol, e.g. `<PRP>`.

3.6.2 Token, Sentence, Article

Tokens are the central unit in developing a way of representing articles as finite-state automata. We shall trace this process from a trivial starting point to the final solution. If we take **(13)** as our example sentence in (simplified) COSA XML, **(14)** would be a naive SFST-PL rendition, which we could compile into an FSA.[37]

(13)
```
<sentence id="sample-1">
  <token id="sample-1-1" lemma="Tony" POS="NNP">Tony</token>
  <token id="sample-1-2" lemma="Blair" POS="NNP">Blair</token>
  <token id="sample-1-3" lemma="speak" POS="VBZ">speaks</token>
  <token id="sample-1-4" lemma="." POS=".">.</token>
</sentence>
```

(14) `<S> <T> Tony <T> Blair <T> speaks <T> \.`

[36] A corresponding file for the Penn Treebank Tagset is included, but others can be added easily.
[37] Remember that spaces are irrelevant. Also, the period needs to be escaped because of its special meaning in SFST-PL (and regular expressions in general).

One of the problems with **(14)** is that, while using a minimal number of delimiters, there is no single way to recognize the end of a token, since the next symbol could be `<T>`, `<S>` or indeed "nothing". This is why we use additional delimiters to unequivocally mark both the beginnings and ends of tokens and sentences (`<|->`, `<-|>` and `<((>`, `<))>`, respectively):

(15)
```
<((> <|-> Tony <-|> <|-> Blair <-|> \
<|-> speaks <-|> <|-> \. <-|> <))>
```

It might look like the expression is overloaded with delimiters now, but as we add more information to the token, this will make better sense. The next issue has to do with the results we hope to get out of the annotated version of our article FSA. Remember that we ultimately want to assign token IDs to entity IDs. If we used the present scheme to string together our tokens and sentences, it would be quite tedious to find out the position of a token in the article; we would have to count the number of preceding sentences and tokens. Finite-state devices are, generally speaking, no good at counting, so we will add two numbers to each token; one for the sentence and one for the token count:[38]

(16)
```
<((> <|-> <1><1> Tony <-|> <|-> <1><2> Blair <-|> \
<|-> <1><3> speaks <-|> <|-> <1><4> \. <-|> <))>
```

An FSA as it is defined by **(16)** allows efficient matching and uniquely identifies each token so we can later map the position to a COSAtoken ID. But it does not yet contain all the information that could be useful. We will see that POS tags help to correctly recognize political actors by name, so we add them to the FSA. Moreover, we have to remember our concept of annotation. We have said that the entity FSTs, which will be composed with the article FSA at hand, will add entity IDs to the tokens that are recognized as being part of an entity. These entity IDs will consist of "free text" (in other words: `$Text$`). If we want to add them next to the token's word form, we need another delimiter (`<+>`) to set them off. As a final improvement, the use of FSAs to represent the article's text allows us to include both the word as it appears in the text as well as the lemmatized form at almost no cost, using a union (`|`).[39] Tests have shown that because of

[38] See the discussion above for the reason to use number symbols rather than plain numbers or even token IDs.

[39] The example has a more concise representation, of course (`speaks?`), but the result after compilation and minimization is equivalent.

incorrect (manual) lemmatization in the entity lists supplied, this can lead to better results. Eventually, the FST-PL representation of our example sentence looks like this:

(17)
```
<((> <|-> <1><1> <NNP> <+> Tony             <+> <-|> \
     <|-> <1><2> <NNP> <+> Blair            <+> <-|> \
     <|-> <1><3> <VBZ> <+> (speak | speaks) <+> <-|> \
     <|-> <1><4> <.>   <+> \.               <+> <-|> <))>
```

The FST-PL that ERCOSA constructs from XML articles looks exactly like **(17)**. The automaton that results from its compilation has all the information required to recognize political entities and allows for transparent rule writing and efficient processing. The formal definitions for a token, a sentence and an article are the following:

(18)
```
$Token$   = <|-> [#Nr#] [#Nr#] [#POS#] <+> $Text$ <+> \
            $Entity$* <-|>
```

(19)
```
$Sent$    = <((> $Token$+ <))>
```

(20)
```
$Article$ = $Sent$+
```

We have not, so far, mentioned `$Entity$*`, which appears in **(18)**. It reserves a slot for nil, one or several entity IDs assigned to the token. As we go on to discuss entity FSTs which will be "applied to" article FSAs of the form we have just seen, we will return to this part. Once a good scheme for representing the input text has been found, the task of transforming XML articles into FSAs is both fast and fairly straight-forward in terms of the necessary Python code. The transformation of entity definitions into entity FSTs presents a much greater challenge.

3.7 Entity FSTs

The finite-state automata which result from the serialization of the XML newspaper articles into a custom representation, as it was outlined in the previous section, present the basis for the annotation of political entities with the help of transducers. We need to differentiate between two classes of entities for this purpose: there are those defined by their first and last names, and those defined by one or more keyword constructs. The two classes do not entirely coincide

with the *actor* and *topic* entity types, since political parties are of the *actor* type but belong to the *keyword* class of entities. It is obvious that the rules for name matching will differ from those for keyword matching. Hence, we need two different algorithms for transforming the corresponding entity definitions into FSTs. We shall not be concerned with the details of how FST-PL is constructed from an entity record as it appears in the list of entities.[40] The corresponding code is, unfortunately, among the most convoluted in erCOSA. As with article FSAs, we will instead look at the result: in this case, rules for matching and marking the entities.

3.7.1 Name tagging

Our sample entity for the explanation of annotating name entities shall be Tony Blair, who has the following entity record:

(21) `act-6009-50042‖surname=Blair‖forename=Tony`

With only a single first and last name, and no alternative name spellings, this is a basic case rather than one which exemplifies all possible complexities. The principle remains the same, however, and more intricate cases will be mentioned where appropriate.

One might say that our entity FSTs need to perform two tasks: firstly, they have to identify the tokens which represent the entity in question (if there are any) and secondly, they must mark these tokens with the respective entity ID. The latter might be called the central transduction, and in our example, is expressed by the following FST-PL line:

(22) `$X$ = <>:{\<act\-6009\-50042\=N\>}`

This "mini-transducer" by the name `$X$` relates the empty string (ε) to the string "<act-6009-50042=N>".[41] As you can see, this string is the entity ID taken from the entity record (with "=N" appended) enclosed in angle brackets. The suffix serves to retain the information that this entity was found by name matching rules. We now have the means to insert this entity marker in our article, but we

[40] While a name record always results in a single FST, there is in fact one transducer for every alternative definition of a keyword record.

[41] Note the use of backslashes to escape characters, especially "<" and ">". It means that we really are dealing with a sequence of characters rather than a single multi-character symbol.

still need to make sure it is inserted where it belongs, and nowhere but there. Remember that the formal definition of a token included an variable for this purpose, which has the following definition:

(23) `$Entity$ = \< "<entity_id.a>" \= [NK] \>`

Again, we can see the enclosing brackets and the suffix (which could also be "=K" for keyword matches). The middle part (`"<entity_id.a>"`) dynamically includes a pre-compiled FSA. This is because the ID scheme might change and can therefore be defined in ERCOSA's configuration file. It is `[a-z0-9-]+` at present.

Placing the entity marker in the right spot within the token is, however, not the real challenge; it is marking the correct tokens. This is why we define the following transducers:

(24)
```
$1$  = <|-> [#Nr#] [#Nr#] [#POS#] <+> Tony <+> \
       <>:<F> $X$ $Entity$* <-|>
```

(25)
```
$2$  = <|-> [#Nr#] [#Nr#] [#POS#] <+> Blair <+> \
       $X$ $Entity$* <-|>
```

(26)
```
$-2$ = <|-> [#Nr#] [#Nr#] [#POS#] <+> Blair <+> \
       ^_$X$ $Entity$* <-|>
```

If we compare these to the generic case of a token (`$Token$`), we notice that the textual content of the token is now explicitly defined. Furthermore, **(24)** appends a symbol (`<F>`), marking the token as a first name, as well as the entity marker. `$Entity$*` accounts for the possibility that this token might have been marked as belonging to another entity already. Multiple assignments are thus possible and all entity markers are retained. **(25)** is the same, except for the lack of the first name marker. **(26)** is nearly identical to **(24)**, but the transducer which inserts the entity marker (`$X$`) is preceded by an operator (`^_`) which swaps its two levels, effectively inverting its operation. **(25)** therefore adds an entity tag and **(26)** removes that same tag.

The three name-matching transducers work with single tokens, but with a slightly more complex case (and there are many of those), a name could be made up of several tokens. For this reason, and because we want to be able to refer to entire first and last names irrespective of the number of tokens, we define the following variables.

(27)
```
$First$     = $1$
$Last$      = $2$
$-Last$     = $-2$
```

(28)
```
$First$     = $1$ | $2$ | $3$ $4$ | $5$
$Last$      = $6$ | $7$ $8$ | $9$
$-Last$     = $-6$ | $-7$ $-8$ | $-9$
```

The case of Tony Blair is trivial (**(27)**), but the case of Mr Wedgwood(-Benn) (**(28)**, compare with **(6)** on page 25) shows that these variables are in fact useful. They allow us to refer to all of the 15 possible forms of referring to this politician in a very concise manner.

In order to tag only those occurrences of the actor's name tokens which do actually refer to Tony Blair, we define two more auxiliary variables:

(29)
```
$Self$      = $First$ $Last$
$Other$     = "<anyfirst.a>" _$Last$
```

The first caters for the uncomplicated case where we have a first name variation immediately followed by a last name variation. In that situation, we always assume that the article does indeed refer to our politician.[42] Note that `$Self$` defines a transducer; it will not only match the respective tokens but also insert the entity tag. The second variable (`$Other$`) is of central importance for erCOSA's solution to resolving the most prominent ambiguity in name matching: who does a last name which is not accompanied by a (known) first name refer to? There are, for example, four different politicians by the (family) name of Brown in the *uk_2005* entity set: Gordon, Nick, Michael and Marilyn. For the discussion of this question, let us suppose that before referring to people by their last name only, they will have been introduced with their full name.[43] One way of tackling the problem, hence, would be to assign last names to whichever actor by that name was last "seen" with their full name. There are a few problems with this approach:

1. *Ulterior decision*

   Our entity FSTs know nothing of each other, and because they are processed

[42] Even full names are not unique, of course, but detecting this type of ambiguity is beyond the current goals for erCOSA.

[43] While not an unreasonable presumption, in the case of articles there will often be a shorthand reference to people in the article's title. The ambiguity will go unresolved in that case.

serially, there is no telling if another entity might want to "claim" a given last name token later on.[44]

2. *Hidden coreference*
   The textual content of the article itself does not always allow finding a previous occurrence of the same person. Consider a text which introduces "Anthony Wedgwood-Benn" and later refers to him as "(Mr) Benn"; there are no identical tokens here, and yet there is a coreference. A given entity FST "knows" about variations of its own actor's names, but not that of others. Finite-state devices are not, overall, well suited for algorithms which rely on a global memory.

3. *False positives*
   Consider the following (fictional) text:

   The London premiere of The Da Vinci Code*, the season's film to beat at the box office, was attended by Gordon Brown, who admittedly takes an interest in the writing of his namesake: "You would think that daily politics offers plenty of conspiracy theories, but I read most of Dan Brown's books." Brown is the American star novelist whose book has been adapted by director Ron Howard.*

   This example shows that we cannot only take into consideration the names we know; we have to expect references to people who are not on our list but share their family name with politicians who are. We cannot look at the token "Brown" in the second sentence as a last name on its own. And we should take Dan Brown into consideration when the name "Brown" does, later in the article, appear on its own. The example is fabricated and unlikely, but in a longer article of some hundred sentences, a shift from one to another person by the same family name is not unlikely to happen.

Consequently, we want a name matching scheme which has no need to know about other actors on the list, or what tokens the corresponding FSTs might tag. It should also respect unknown combinations of first and last names as unknown persons and avoid tagging them. `$Other$` is the stepping stone to achieving these

[44] If this were the only issue, the implementation of a second-order resolver for this kind of ambiguity would indeed be preferable.

goals. You may have noticed that it includes another automaton, *anyfirst.a*, which is compiled from the following FST-PL expression:

(30) `<|-> [#Nr#] [#Nr#] [#NamePOS#] <+> $Text$ <+> $Entity$* <-|>`

It is evident, now, why we elected to include POS tags in our article FSA. We could have tried to recognize "unknown" first names with the help of a list of the most frequent ones. However, relying on POS tags instead is not only language-independent (as far as erCOSA is concerned), but tagging also seems the more appropriate stage of processing to decide such matters. For English, at least, names were tagged very accurately.[45] As you can see, *anyfirst.a* will match any token which has a `#NamePOS#` tag, a set of tags which is configurable. Importantly, POS tags are only used to recognize unknown persons, i.e. if relevant politicians appear with their full names, they will always be tagged, irrespective of potential tagging errors. Notice, too, that while `$Self$` is a transducer which changes its input by inserting entity tags, `$Other$` is an automaton which merely matches tokens.[46] As a final detail, `$Other$` would seem to also match `$Self$`, but it does not. This is because the latter inserts the `<F>` marker into the first name token, which afterwards no longer matches *anyfirst.a* or indeed the generic definition of a token. We thus have to make sure we "apply" the transducer `$Self$` to the article FSA before trying to match `$Other$`.

So far, we have defined automata and transducers which allow us to match and tag relevant tokens. If the (input) string does not match, however, they will produce an empty string, which is clearly not what we want. Eventually, we should therefore have one or more transducers which make use of the replacement operator mentioned earlier (`^->` in FST-PL). It constructs FSTs which "return" an unmodified input string in case the replacement does not apply. This is the syntax of replacement rules:

(31) *transducer* `^->` *left_context* `__` *right_context*

[45] It seems reasonable to suppose that statistical taggers perform better at recognizing proper names than consulting a list, which will always miss more exotic first names. There were no tests to that end, however.

[46] The underscore in `_$Last$` selects only the "upper" language (XFST terminology), which, unfortunately is called "deep" (or "analysis") language in SFST. The two programs basically have an inverted perspective on the two levels of a transducer and erCOSA uses SFST with the XFST perspective. Which, in a way, makes no sense at all, but in the end, is luckily irrelevant...

Each string in the input language of the transducer on the left hand side is mapped to its transduction in case it appears in the input language of the transducer defined by this expression with matching left and right contexts. The contexts must be defined by automata and not transducers. We always have to define at least one context, which can be the empty string ε (<> in FST-PL). The following definitions show how ERCOSA uses replacement rules for name matching; each rule is accompanied by a paraphrase.

(32) `$TagFull$ = $Self$ ^-> <> __`

If a first name and a last name variant appear next to each other, tag the corresponding token(s), irrespective of the context.

(33) `$TagLast$ = (<((> | "<nofirst.a>") $Last$ ^-> <> __`

If a last name variant stands at the beginning of a sentence or follows a token which is not a proper name, tag the corresponding token(s), irrespective of the context.

(34) `$PruneLast$ = $-Last$ ^-> $Other$ !(.* <F> .*) __`

Remove the entity tag from the token(s) of a last name variant if anywhere in the left context, there is another person by the same last name with no occurrence of the full name in between.

(35) `$Cleanup$ = <F>:<> ^-> <> __`

Remove any occurrence of the `<F>` *marker, irrespective of its context.*

(32) is the straight-forward case of tagging a full name. Note that after passing an article FSA through this transducer, all first name tokens that appear as part of our politician's full name contain the `<F>` marker. **(33)** should tag all of our politician's last name variants that occur on their own. The included FSA *nofirst.a* is identical to *anyfirst.a* except that the matched token must *not* have a proper name POS tag. Normally, there will be no overlap between **(32)** and **(33)**, avoiding double tags.[47] **(34)** performs the important task of correcting the "over-confidence" of **(33)**, which assumes that all stray last name variants of the actor in question do necessarily refer to just that actor. `$PruneLast$` is the expression which is responsible for the high degree of complexity of entity FSTs in ERCOSA. The problem is with the left context of the replacement rule. The expression

[47] More than one entity tag per token is not a problem, however, and if they are the same, they will be reduced later on.

`!(.* <F> .*)` describes the language which includes every possible string (over the defined alphabet) that does not contain `<F>`. Forming the complement of such an FSA is a computationally expensive operation. In other terms, it means that the scope of this rule always extends back to the beginning of the article, whereas all the other replacements are local in that they are concerned with no more than a few subsequent tokens at a time. **(35)**, finally, removes all first name markers from the article. They are meant to be transitory and are only significant with respect to the entity at hand.

There is often more than one way of writing replacement rules. For example, there is no logical drawback to rewriting 33 in this way:

(36) `$TagLast$  = $Last$ ^-> (<((> | "<nofirst.a>") __`

In cases like this, it makes sense to check whether one of the alternatives is more efficient in terms of the processing time they require. The expression in question, however, is pretty unproblematic anyway. With **(34)**, any improvement would be very welcome, and one might think that the following alternative is viable:

(37) `$PruneLast$ = $Other$ !(.* <F> .*) $-Last$ ^-> <> __`

This was tested and did turn out to be slower, possibly because `$Other$ !(.* <F> .*)` will match once for each `$Other$` preceding a given `$-Last$`. With this way of writing the rule, every match would be processed, even though the result will always be the same.

Our example name-tagging entity FST only consists of variable definitions so far. The final expression of every FST-PL document must be an expression which uses these variables but must not be assigned to a name itself. This expression then defines the FST as it results from the compilation of the entire FST-PL document. Replacement rules are usually cascaded using composition (i.e. the output of the first is the input of the second, and so on), and the order for our example should come as no surprise:

(38) `$TagFull$ || $TagLast$ || $PruneLast$ || $Cleanup$`

We first tag full names and all "lonely" last names. We then remove the spurious last name tags and finally the first name markers. After "passing through" this entity FST, the article has been annotated for the respective entity and is ready

for another entity FST. It might be another name-tagging device, or indeed a keyword-tagging transducer, which we will look at next.

3.7.2 Keyword tagging

The principles of ERCOSA's keyword tagging are the same as those of name tagging. We define the tokens which are of interest for a given entity, check whether they appear as required by the entity definition and assign the respective entity tags. The specifics differ, however. There is no differentiation between two kinds of tokens (first and last names), and we always match an entity in the context of an entire sentence. The challenge of ambiguous last names disappears, but instead we have more complex entity definition constructs and must correctly implement the three keyword operators (see Table 3.1 on page 28).

As with name-tagging FSTs, a running example will be used to explain the FST-PL code that compiles into the actual transducer. Unlike the real-world *Tony Blair* entity we used above, the keyword entity is an artificial one, which makes use of all the operators and thus enables a thorough understanding of the implementation:

(39) `top-0‖keyword=alpha beta & gamma &!  foo bar`

If, within a single sentence, the tokens "alpha" and "beta" occur in succession as well as the token "gamma", but "foo" and "bar" do not occur in succession, then tag the respective tokens with "<top-0>".

The paraphrase outlines our task, and we will start with the last part, the insertion of the entity tag:

(40)
```
$X$ = <>:{\<top\-0\=K\>}
```

As you can see, the transducer which handles the tag itself is exactly the same as with the name entities, except that we add a trailing "=K", which marks the entity as one that was defined by keywords. As with names before, we also need to define the tokens that are of interest:

(41)
```
$1$ = <|-> [#Nr#] [#Nr#] [#POS#] <+> alpha <+> \
      $X$ $Entity$* <-|>
$2$ = <|-> [#Nr#] [#Nr#] [#POS#] <+> beta  <+> \
      $X$ $Entity$* <-|>
$3$ = <|-> [#Nr#] [#Nr#] [#POS#] <+> gamma <+> \
```

```
      $X$ $Entity$* <-|>
$4$ = <|-> [#Nr#] [#Nr#] [#POS#] <+> foo   <+> \
      $X$ $Entity$* <-|>
$5$ = <|-> [#Nr#] [#Nr#] [#POS#] <+> bar   <+> \
      $X$ $Entity$* <-|>
```

There is nothing spectacular about these either. Note that we will not need the functionality of removing a previously added tag, so we need only one definition for each token.

Where we assembled a definition for all first and all last name variants before, we now group the tokens according to whether they need to follow each other:

(42)
```
$Tag1$  =  $1$ $2$ ^-> <> __
$Tag2$  =  $3$ ^-> <> __
$Tag3$  =  $4$ $5$ ^-> <> __
```

The five tokens have thus been reduced to three token groups, "alpha beta", "gamma" and "foo bar". It is important not to misread the variable names. There are not three different tags; rather, these are three transducer definitions which all insert the same tag on different token groups. You may notice that `$Tag3$` is useless, since by definition of our entity, "foo bar" will never be part of it and hence will never receive the tag. While it would be possible to omit unnecessary variable definitions, the automatic generation of FST-PL from entity definitions was kept simple. Since the gain in efficiency would be fractional, the effort necessary to avoid these lines could well compensate the savings and is hence not worthwhile.

In the previous section, we have seen that transducers are not suitable for all purposes. But it is not only in the left and right context of a replacement rule that automata are needed instead. There are also finite-state operations under which FSAs are "closed" but FSTs are not.[48] This is true for intersection, at least if we take the general case of an FST which is not epsilon-free (Roche and Schabes, 1997). But intersection is precisely the operation we need, because we are looking for a number of token groups, all of which have to occur (or indeed not occur) in one and the same sentence to actually represent our entity. We could instead try to enumerate all the permutations in which the significant tokens could occur in a sentence, but that quickly becomes tedious. What we need,

[48] Finite-state devices (i.e. the regular languages or relations they represent) are closed under an operation if the result is, again, a finite-state device (i.e. a regular language or relation).

therefore, is a number of automata which we can intersect to give us an FSA that matches all sentences in which tokens are present or absent as required by the entity definition.

(43)
```
$Find1$ = ($Token$* _$1$ _$2$)+ $Token$*
$Find2$ = ($Token$* _$3$)+ $Token$*
$Find3$ = ($Token$* _$4$ _$5$)+ $Token$*
```

Remember that `$1$` to `$5$` are transducers, but because we need automata and also do not want to insert any tags at this stage, we use them with the operator which "extracts" the upper level (`_`). The use of `$Token$*` on both sides of the expression makes sure that an entire sentence will be matched. This is necessary for the intersection to work as expected: all expressions need to have the same "alignment" within the article. A *Kleene plus* (a variation of the *Kleene star*, meaning "one or more") is appended to the part before the final `$Token$*` to make sure that the expression will match a sentence only once, even if that sentence contains multiple occurrences of the token group in question. Note that there is no sign of negation in `$Find3$` as of yet, but we know that in order for a sentence to match our entity, it must *not* contain those tokens. The following expression takes care of that as well, and defines the FSA which will match any sentence which contains our entity as per its definition:

(44)
```
$Find$ = $Find1$ & $Find2$ &! $Find3$
```

If we compare this FSA definition to the entity definition of **(39)**, we can see that the operators are exactly the same. In FST-PL, the ampersand (`&`) signifies an intersection and the exclamation mark (`!`) forms the complement of the expression that follows it. Complements can be bothersome in that they match absolutely everything except the language defined by the original automaton. In the case of `!$Find3$`, we need to be aware that it will match strings of any length and shape, from a single character or symbol (e.g. "I" or "<6>") to entire articles. The only thing it will not match is a sentence (excluding the delimiters `<((>` and `<))>`) that contains the tokens "foo" and "bar" (in their respective ERCOSA-style representation). It will, in other words, return an enormous amount of alternative matches for an article. It is important, then, to always intersect such an expression with another, more "positive" one, to keep it "under control". A valid COSA entity definition construct guarantees at least one "positive" token group,

so we are on the safe side here.

`$Find$` will find us sentences in which all the right tokens are either present or absent, but since it is an automaton (also called *acceptor*), it cannot mark tokens, and we have no means of knowing where the relevant tokens are. For this reason, we still need an expression that actually inserts the entity tags, once we have found a suitable sentence:

(45) `$Tag$ = $Tag1$ || $Tag2$ || $Tag3$`

The transducers which tag the individual token groups are simply cascaded together. On its own, `$Tag$` would tag every instance of those token groups, so it needs to be combined with `$Find$`.

(46) `($Find$ || $Tag$) ^-> <((> __ <))>`

This expression is not assigned to a variable; it therefore builds the final keyword-tagging entity FST. What happens here is that `$Tag$` is applied to every sentence that matches `$Find$`. The FST that results from this composition is the left hand side of a replacement rule. This is necessary output an unchanged article FSA in case no sentence matches. The sentence boundary symbols as left and right contexts shorten the processing time of the transducer, possibly because they provide unequivocal "anchors".

While some parts of the FST-PL document that defines a keyword-tagging entity FST are almost identical to that of its name-tagging counterpart, there are some major differences. With name tagging, the complexity lies in avoiding the "overconfident" tagging of last names. With keyword tagging, it is the entity definitions themselves which ask for a complex matching algorithm. Most entity definitions are not as complex as the one in our example, however. On average, the time it takes to tag a keyword entity multiplies for name entities. It should also be noted that while every name entity generates a single entity FST, each keyword construct that is defined for an entity generates its own FST. We can see, now, how sequentially applying every entity FST to the article FSA will lead to a fully tagged article. It is still represented as a finite-state device, which generates strings as per our earlier scheme of representing tokens and sentences. The next section will be concerned with the extraction and further processing of the results.

3.8 Result processing

Once the FSA that represents a COSA newspaper article has been composed with all the entity FSTs, which are based on the list of relevant entities, we are still dealing with a finite-state transducer. We are only interested in the output language of that transducer and can hence discard the other level and view it as an automaton. All the information we need is "in there", but we have to devise a way of extracting and collating the results into a format that is useful. Based on these results, we will have to modify the XML representation of the article and insert the found entities as appropriate.

SFST has a method for retrieving all the strings that an FSA generates. While the tagging scheme we use should never lead to an infinitely ambiguous FSA, the number of output variations for an article is easily in the thousands. Since we allow more than one entity tag on a token, the reason for this are neither unresolved name ambiguities nor tokens that are part of more than one keyword entity. The variations exist because we allow more than one word form for each token (compare **(17)** on page 43), i.e. the inflected form of the word along with its lemma. Whenever there are alternatives, only one form of each token is chosen for a single output string. The number of variations therefore multiply with every additional disjunction. If we do not want to process countless strings of great length, we need to reduce the complexity of our tagged article automaton. For this purpose, we first apply a "cleaning" transducer.

(47) `$0$ = [#Char#]:<> ^-> <+> [#Char#]* __ [#Char#]* <+> <-|>`

(48) `$1$ = {[#Nr#][#Nr#]}:<> [#POS#]:<> ^-> __ <+> <+>`

(49) `$2$ = {<|-><+><+><-|>}:<> ^-> <> __`

(50) `$0$ || $1$ || $2$`

Like entity FSTs, it is defined by a number of replacement rules which are eventually composed. In **(47)**, all characters that appear in between the text delimiters (`<+>`) in a token that has not been tagged (i.e. the end-of-token symbol follows immediately) are replaced by the empty string.[49] The rule in **(48)** goes on to

[49] The expression might seem overly complicated. However, `$Text$:<>` is not a valid FST-PL construct, because `$Text$` is itself an automaton. Only strings or character classes (like `[#Char#]`) can be used when defining a transduction of this kind.

delete the sentence and token number symbols as well as the POS tag from untagged tokens. Finally, **(49)** removes the empty token "skeleton" altogether. **(50)** is the composition of the individual rules in just that order.

The transducer which results from removing all untagged tokens is much smaller, and the output strings it generates much shorter. The sentence and token numbers in each token allow us to still correctly identify the tokens, even though entire sentences may have been stripped out. The strings generated by this "cleansed" transducer do not yet lend themselves to further processing easily, though. There are still many output variations, because many of the tagged tokens themselves will have a lemma that differs from the inflected form. All entity tags will have been assigned to only one form, even though both forms could potentially have received a (differing) set of tags.[50] If we ignored the possibility that both forms of a word can receive a different tag (one of which would probably be unintended), we could try to find the output variation with the most tokens (in itself not a straight-forward task, because the longest string does not necessarily have the most tokens) and ignore all others.[51] Even then, it would be a laborious task to parse the output string and collect the information we need. Luckily, finite-state devices are very good at parsing and collating. For the final time, then, we make good use of the finite-state formalism to best prepare the entity recognition results for onward processing.

The first expression of this result extraction device effectively chops up the article into token units:

(51)
```
$1$ = ([#Char# #POS# #Delim# #Nr#]:<>)* \
      $Token$ \
      ([#Char# #POS# #Delim# #Nr#]:<>)*
```

The first and third lines in **(51)** will map any number of symbols or characters to the empty string; they delete anything.[52] What remains is the single `$Token$` in between, which will of course match every single token of the article in turn.

[50] In the process of stripping all untagged tokens, we could have removed all `$Text$`, even from the tokens which are tagged with an entity. It is true that the actual word that was tagged is no longer essential. There are advantages to keeping it, though: firstly, it is very helpful for debugging, and secondly, it later allows us to make sure the sentence and token numbers are "in sync" with the XML article, i.e. we really are tagging the right token.

[51] This is because all combination of word forms will have been generated by the previous transducer, hence one of them will contain the superset of all assigned tags.

[52] It is not possible to write this as `.:<>`, because in SFST, this would require all the intended mappings to be part of the alphabet.

Note that this transducer is not formed by a replacement rule and its output is an FSA which generates every (tagged) token of the article on its own. This also means that instead of many different combinations of tokens strung together, as previously, we have now reduced our data to just the individual tokens without any redundancies. A single output might look like this:

(52)
```
<|-><1><1><NNP><+>Brown<+><act-6009-50066=N>
<act-6009-50063=N><-|>
```

You will notice that the token has two entity tags. This might be because the name appears at the beginning of the article, without an accompanying first name, and cannot be resolved to just one actor. This output is very useful already, but to make the transition from finite-state devices back into the realm of Python even more comfortable, we have another transduction in place:

(53)
```
$2$ = <|->:<> [#Nr#] <>:{\9} [#Nr#] [#POS#]:<> \
      <+>:{\9} $Text$ <+>:{\9} \
      ([#Char#]:<>)* \
      {\<}:<> "<entity_id.a>" {\=}:<> <>:{\9} [NK] {\>}:<> \
      ([#Char#]:<>)* <-|>:<>
```

On the one hand, there is some formatting: tabulators (`\9`) are inserted, the POS tag and some delimiters are deleted. On the other hand, characters to the left and the right of an `"<entity_id.a>"` are removed and the entity type that was appended to the ID is separated. This means, in other words, that similar to the expression above, every entity tag (if there is more than one) will be matched in turn. If we were to apply the transducer defined in **(53)** to **(52)**, we would get two output strings ("‖" signifies a tab stop):

(54)
```
<1>‖<1>‖Brown‖act-6009-50066‖N
<1>‖<1>‖Brown‖act-6009-50063‖N
```

pySFST returns alternative output strings in a list. Conveniently, we therefore end up with a list of tab-separated records that contain only relevant information and are consistently formatted. These records all relate a single token to a single entity ID. If we know how COSA specifies the structure of token IDs, we can combine the sentence and token number and instead speak of a token ID:

(55) *token_id [1]* → *[1] entity_id*

The question that we would normally ask, however is not *Which are the tokens that make up entities, and what entity was assigned to each of them?* but rather *Which entities were found, and by what tokens is each of them manifest in the text?*. The results we get from the finite-state processing need to be collated and assembled into a different relation:

(56) *entity_id [1] → [N] token_ids*

Because of the way in which we use finite-state transducers to match and tag the entities, we cannot really tell which of the tokens sharing the same tag were recognized as a single instance of the entity. In the case of names, consider the following example:

(57) *We had a chance to talk to Tony Blair, Blair being the first to arrive.*

There are two occurrences of the same actor in this sentence, "Tony Blair" and "Blair". They are separated by a token (the comma), and it is very unlikely that two instances of the same actor would ever follow each other immediately. Hence, we can use the following rule for the assembly of name entities:

(58) *All successive tokens which were assigned the same entity ID by the name tagging scheme belong to the same entity.*

The current specification of keyword entities is explicit about the scope of an entity. Even if it might be undesirable, a keyword entity always spans the entire sentence. Consider an entity which requires "healthcare" and "regulation" to co-occur within the sentence.

(59) *Asked about the government's stance on the increase of healthcare costs, Brown said that further regulations across Britain's healthcare system are being reviewed.*

(60) *Healthcare regulations abound: many voice their anxieties about the effects of the government's planned regulations in the healthcare sector.*

Even though we would argue that the entity occurs only once in **(59)** but twice in **(60)**, ERCOSA will recognize it only once in either sentence and assign it three and four tokens, respectively. Until keyword entities are defined differently (e.g. using smaller syntactic scopes or more specific syntactic relations between the

tokens), this can hardly be avoided. The collation rule for keyword entities is obvious by now:

(61) *All tokens within a sentence that were assigned the same entity ID by the keyword tagging scheme belong to the same entity.*

Because the criteria for deciding which tokens belong together differ for name and keyword entities, we have retained a mark of which tagging scheme was used ("N" or "K", compare **(54)**). Having correctly assembled all found entities, we still need to transform them into XML, which is then merged with the original article document.

COSA specifies what a found entity should look like in XML, and we can easily construct the required nodes using Python's DOM-style manipulation facilities. Returning to the sample sentence in **(13)** on page 41, erCOSA would generate the following XML:

(62)
```
<entities>
  <actor listid="act-6009-50042">
    <tokenref ref="sample-1-1"/>
    <tokenref ref="sample-1-2"/>
  </actor>
</entities>
```

This subtree gets added to the corresponding sentence node in the original XML. Once all found entities have been collated, transformed into XML and added to the article document, erCOSA has done its work.

Over the last few sections, we have seen how a given newspaper article, as it is output by preCOSA, is transferred into the domain of finite-state processing, where tokens that make up relevant entities are matched and tagged. In the end, the tagged tokens are collected from the resulting finite-state device and assembled into found entities, which are added back into the initial XML document. The focus has been on linguistic issues and the finite-state algorithms used to address them. But there is another side to erCOSA: in aiming to recognize hundreds of entities in hundreds of articles in a robust and user-friendly fashion, it faces challenges in the domain of engineering, too.

3.9 Technical issues

The technical challenges that come with the implementation of an application that is to be put to productive use are usually defined by a number of restrictions: limited space (in memory, on disk), limited time (CPU, I/O) and last but not least, a limit to the complexities and constraints that users are willing to put up with. These limits are often closer than one might think, and all of them were threatened by erCOSA. In this section, we shall look at the reasons for these issues as well as the steps that were taken to confine them.

3.9.1 Footprint, speed and workflow

The size that is taken up by the relevant data for a particular task, the time it takes to perform the necessary operations with that data, and the assignment of these operations to individual tasks are all interrelated. We prefer to keep large pieces of data on the hard disk rather than in memory, because memory is always a precious and scant resource. If a piece of data is frequently used by a particular operation, however, we prefer to have it readily available in memory, because reading it from the disk takes a lot more time. If an operation takes a lot of time, finally, we prefer to keep it in a separate task that is executed only when necessary, and ideally saves its result persistently, i.e. on the hard disk. All of this is relevant for erCOSA. In order to understand its sub-division into two main tasks, *update* and *process*, we need to look at some figures first. Table 3.2 lists data sizes and Table 3.3 processing times.

Document	Total number	Format	Total size (MB)	Avg. size (KB)
Article	779	XML	73	94
		FST-PL[a]	n/a	n/a
		binary[b]	63	83
Entity	487	FST-PL	2	4
		binary[c]	534 / 55	1'097 / 113

[a] Articles are not transformed into SFST-PL in their entirety, but sentence by sentence.
[b] These values are extrapolated.
[c] The second number indicates the compressed size.

Table 3.2: erCOSA space usage

Process	Document	Total time (h)[a]	Avg. time (m)[a]
Compilation	Article	0:56	0:04
	Entity	16:20	2:01
Composition		13:51	1:04

[a]All times taken using a single erCOSA instance (which roughly means using only one core) on an AMD Opteron 285 (2.6 GHz) system. The articles and entities are the same as in Table 3.2. The times are extrapolated.

Table 3.3: erCOSA processing times

erCOSA's data falls into two categories: articles and entities, both in their various forms. Articles come as XML files, as automaton descriptions (SFST-PL) and as complied (binary) automata. The number of articles to be processed is not limited, but for the purpose of developing and testing erCOSA, there were 779 of them. Entities come in the form of the supplied entity list, which is so small as to be negligible, as transducer descriptions (SFST-PL) and as compiled (binary) transducers. The number of entities will always be limited, but can vary, of course. For the purpose at hand, there were 487 of them.[53]

In dealing with data size, there are a number of questions we need to have in mind. What data do we need in memory, what can remain on disk? Are the memory and disk footprints affordable? Can we easily optimize them? Looking at articles, we can see that with 94 KB and 83 KB in their XML and binary form, respectively, they are quite manageable in size. But since in total, the articles at hand come to a hefty 94 MB for the XML and 83 MB for the automata, it is probably a bad idea keep all of them in the memory banks, especially since there might just as well be thousands of them. Luckily, there is absolutely no need to do that. We can simply process one article at a time: parse the XML, transform it into SFST-PL and compile the binary finite-state automaton. The last operation is the slowest of the three, but at around four seconds on average (see Table 3.3), it is not worthwhile to cache the result by storing it on the hard disk, especially since an article will not typically be processed more than once.

Before we go on to see what happens with the compiled article, which, so

[53]In actual fact, the list contains 275 name entities and 130 keyword entities, making a total of 405 entities. As we have discussed earlier, keyword entities result in more than one transducer if they are defined by more than one keyword construct. Hence the higher number is used here.

far, is no more than a representation of the input XML, we should examine some non-intuitive behaviour. Rather than compiling all of the SFST-PL that is generated during the transformation from the article XML in one go, erCOSA compiles sentence after sentence and concatenates the resulting automata. The end result is exactly the same. The need for splitting up the compilation into smaller tasks comes from an apparent shortcoming of SFST when dealing with long SFST-PL documents. SFST is meant to be used for morphological analysis (or generation) mainly, where transducer definitions will typically contain just a few rules that work on single words or even letters. The FST-PL that erCOSA uses to compile transducers is in that range as well. In comparison, the FST-PL needed to describe an entire article is absolutely massive. While we still have to think of SFST's crashes as a bug in the program, it is not one that would normally surface. Fortunately, it easily worked around, at only a small speed penalty.

Once we have compiled our article automaton, we also need all the entity FSTs for composition. If the time it takes to compile an article is moderate, the same cannot be said for entities. Table 3.3 lists an average of over two minutes, which, for the set of entities used in this study, makes for a total compilation time of over 16 hours. But time is not the only concern. The average size of a compiled entity FST is close to a megabyte, which adds up to over five hundred megabytes in total. Remember that we need to compose the article with all of the entities. Clearly, there are consequences to be drawn here. We would like to have the entity FSTs available in memory, because we need all of them for every article, but over 500 MB of RAM is too much, surely. Moreover, we do not want to spend the better part of a day compiling entities every time we run erCOSA, let alone every time we process a new article.

The solution to the space issue with entities is compression. SFST does not optimize its transducers for size, and a lot can be gained by applying general-purpose compression. erCOSA uses the open-source *gzip* algorithm in its least effective (hence fastest) mode, which reduces the size to just over a tenth of the original transducer (compare Table 3.2). We can now keep the compressed entity FSTs (at 55 MB) in memory and unzip them as needed, which is a very fast.

The time issue is arguably more serious, and there is no quick fix like compression is in the case of space shortage. The solution must be to make sure we only

compile entities when it is really necessary. This is why erCOSA differentiates between two main tasks. The *update* task is the one that is concerned with compiling entities if necessary, while the *process* task works with whatever previously compiled entities it finds. Compiling an entity "if necessary" sounds easy enough, but it is not trivial. Under which conditions is it necessary to (re)compile an entity?

1. The entity is a new entry on the entity list.
2. The entity's definition has changed.
3. One of the global "includes" has changed, e.g. the set of valid characters (transducer alphabet), the definition of a variable like `$Entity$`, the proper name tag(s), etc.

In case the third condition applies, all entities have to be recompiled. In the other two cases, only the entities in question should be recompiled. erCOSA makes sure to do so by executing these steps:

1. *Collect includes*
   Some are literal (e.g. *chars.fst*), some need to be compiled (e.g. *anyfirst.a*), some depend on the configuration file (e.g. *entity_ id.a*).

2. *Hash includes*
   Compute an *md5* hash over all includes and compare it to the previous one (stored on disk). Save the new hash if it has changed.

3. *(Re)generate entity SFST-PL files*
   For each entity on the list, transform its definition into SFST-PL (see section 3.7). If an according SFST-PL file already exists, compute a hash over both and compare them. If a change has occurred, overwrite the file with the new code.

4. *Compile and store changed entity FSTs*
   If the includes have changed, recompile all entity FST-PL files. Otherwise, compile all entity FST-PL files that are newer than the respective binary file. Compress and store the compiled transducers on disk.

5. *Prune stale entities*
   Remove all entity FSTs which reside on the hard disk but are no longer on the entity list.

Using this scheme for the *update* task makes sure that it can be run at any point (preferably before the *process* task) and always does the necessary work to get all entities up-to-date, but never wastes time on useless recompilation. Figure 3.2 gives an overview of erCOSA's *update* task.

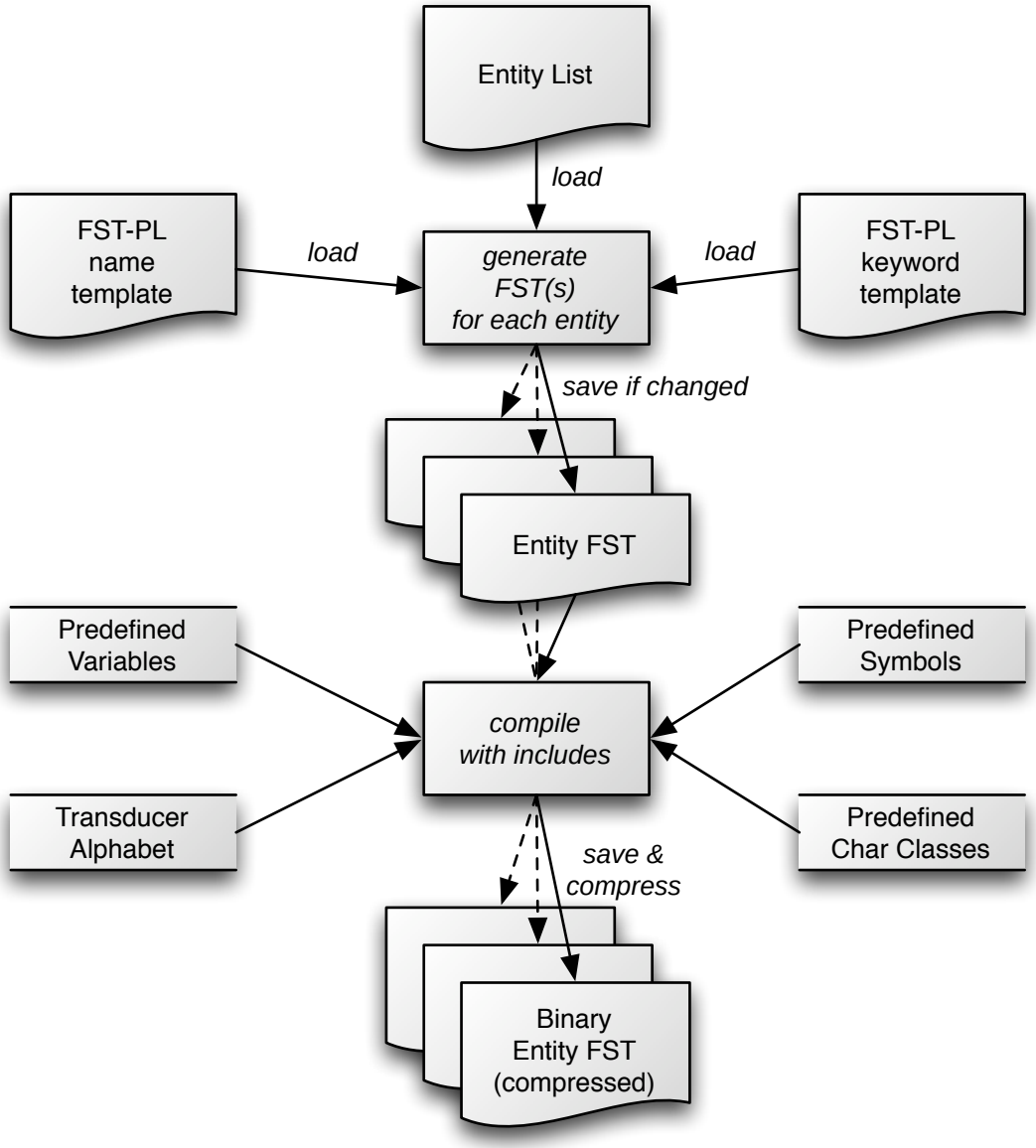

Figure 3.2: erCOSA *Update* Scheme

With entity compilation being taken care of by the *update* task, the *process* task can now focus on the composition of all articles with all entities. As we have

said, it processes one article after the other, in this way:

1. *Load all precompiled entity FSTs from disk*
2. *Parse input article*
   This is the XML file output by preCOSA.
3. *Transform article into SFST-PL*
   See section 3.6 for details.
4. *Compile article FSA*
5. *Compose article FSA with all entity FSTs*
   Each entity FST in turn is decompressed in memory and the result of the composition forms the "input" for the next composition.
6. *Extract results*
   See section 3.8 for details.
7. *Add found entities to XML*

As listed in Table 3.3, the composition of an article with all entities takes quite a long time as well, more than a minute on average. Again, there is not much we can do about this, except avoiding unnecessary work. For this reason, erCOSA only processes an input article if it is newer than the corresponding output article, if it exists. Figure 3.3 gives an overview of erCOSA's *process* task.

3.9.2 Usability and bugs

Very briefly, we shall touch upon the subject of the usability and technical shortcomings and virtues of erCOSA. The bug which prevents us from compiling an entire article in one run has already been mentioned in the previous section. Unfortunately, there is a similar bug which concerns operations like composition on very large transducers. erCOSA implements a crude workaround to avoid crashes: once the article FSA surpasses a critical size, all remaining sentences are being ignored. In the *uk_2005* set of articles, only a single article was suspect to this limitation. Another notable bug, this time with pySFST, was a memory leak which made it impossible to have an instance of erCOSA running for a

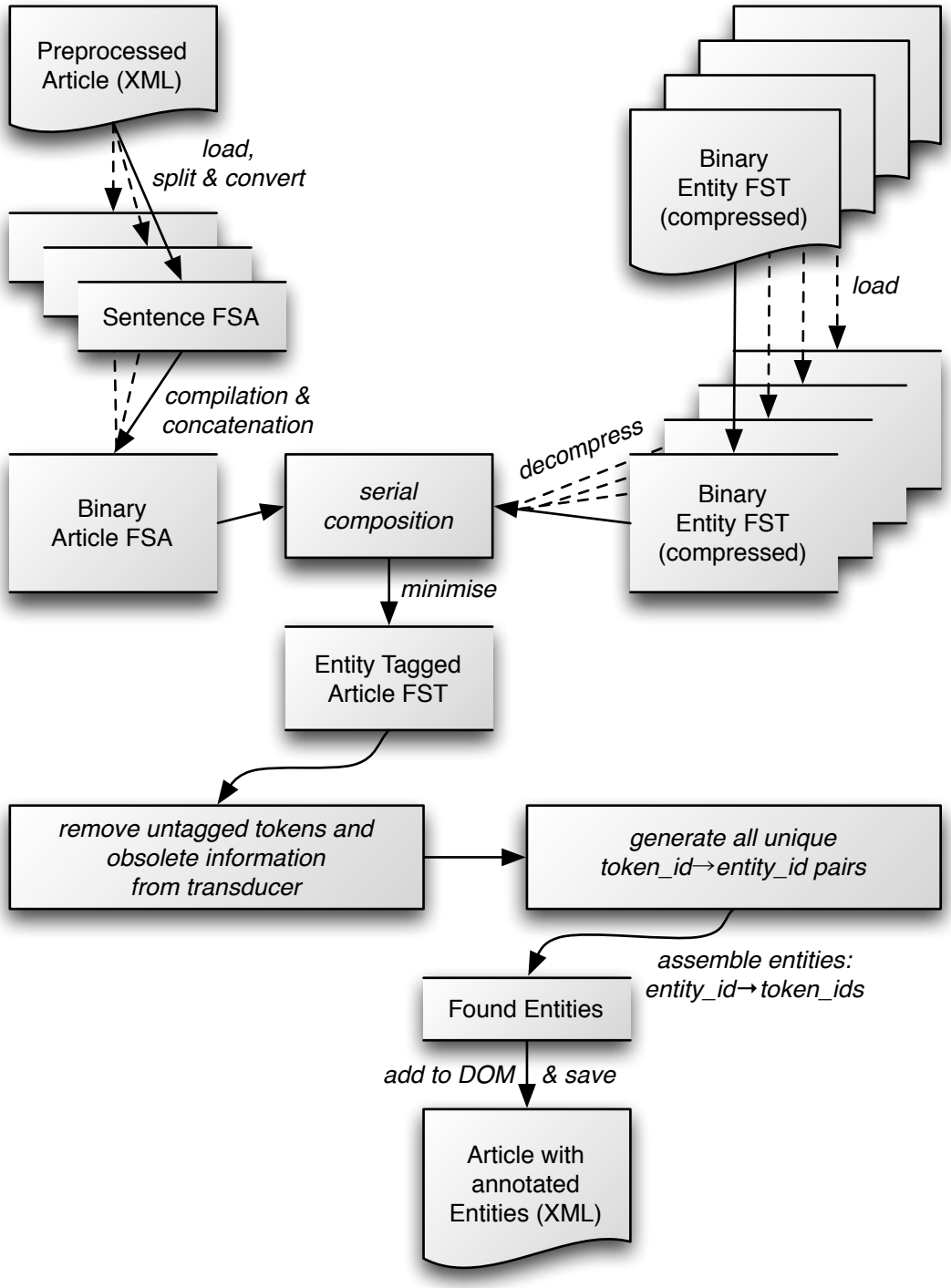

Figure 3.3: erCOSA *Process* Scheme

long period of time and process many articles. Fortunately, a (probably "dirty") fix was found. pySFST also caused segmentation faults under certain conditions related to trying to close a file whose pointer was already discarded. Fortunately, the offending C++ code could be tracked down and corrected.

On the positive side, erCOSA makes sure to be robust and easy to use, while also allowing for enough flexibility and easy debugging. The following points are worth mentioning:

1. *Batch processing*
   erCOSA expects to process many articles with a single invocation; manual arrangements to that end are not necessary.

2. *Easy configuration*
   Every set of articles gets its own configuration file. Only one setting (the base path of the data) is compulsory, but many options can be modified. The configuration file contains an explanation for most options.

3. *Logging and error handling*
   Log messages at various levels are output, both to the console and into a file. The command line option `--debug` leads to more messages and most unexpected program terminations should yield a meaningful error message.

4. *Multiple instances*
   Because one instance of erCOSA will only really use one processor core, and because processing times can be very long, it is desirable to be able to run multiple instances. erCOSA uses a file locking mechanism to make sure concurrent tasks do not interfere with one another. Log messages also indicate the process in question.

5. *Transparency and flexibility*
   erCOSA tries to expose those parts which might need changing. For example, the names of XML tags can be set in the configuration file and the basic layout of entity and name FST-PL files is kept in template files.

6. *Clean directory structure*

7. *Documentation*
   Both the installation and the usage of erCOSA is documented. The con-

figuration file contains explanations for most options and the Python source code has short descriptions for all "public" methods.

More information about the technical aspects of erCOSA can be gathered from the README (see section A.2), the configuration (see section B.1), and the INSTALL (see section C.2) files.

3.10 Assessment

As with preCOSA, no quantitative assessment of erCOSA's recognition performance can be made. The reason, again, is that the *uk_2005* articles have never been manually tagged with the same entities. Hence, there is no reference to check against. Even if there were, though, the figures of such an evaluation would probably say more about the effectiveness of the way in which entities are defined than about the implementation that is erCOSA. We shall thus focus on how well erCOSA solves the problem that is posed by current COSA specifications.

In section 3.3, we compared erCOSA's approach to a previous implementation which did its work using Prolog's unification algorithms. We have seen that the use of finite-state devices is well-established in NLP, but there is not much precedent for the task at hand. erCOSA clearly shows that operating with FSAs and FSTs is a valid method for the kind of entity recognition required by COSA. The challenges that were established on page 29 were all overcome, and erCOSA's results are certainly useful and on-par with previous implementations. It has also become obvious, however, that problems like the resolution of name ambiguities augment the complexity of the involved finite-state devices into the sphere of the barely tolerable. erCOSA has to deal with transducers of many megabytes in size and execution times rise to many hours. Interestingly, Koskenniemi (1997) remarks that just as there are "claims about the inherent simplicity of finite-state solutions", there are those about "inherent complexities in the behaviour of such finite-state systems". He goes on to say that these complexities haven not been observed in the domain of morphology, but posits the following points about the description of syntactic phenomena:

1. "Certain types of rules were difficult to compile into finite-state automata because the resulting machine becomes so large."

2. "The run-time parsing is sometimes either slow or demands a lot of memory or both."

This is very true for erCOSA. In order to arrive at a more efficient implementation, a subgroup of the involved problems should clearly be solved using a different formalism. Even in its current form, though, erCOSA could certainly be faster. There is a lot of room for optimization when defining the rules which make up entity FSTs. As they are now, these rules represent the middle ground between a naive approach and a fully optimized scheme. The consequences of changes in FST-PL with regard to the complexity of the resulting transducer are often non-intuitive and a lot of testing would be required to reach the best overall performance.

Since erCOSA is meant to be used outside of the the Institute of Computational Linguistics and potentially be put to large-scale use, there was a focus on ensuring that it is future-proof and fit for productive usage. As a result, erCOSA does well in the domain of extensibility, flexibility, usability and stability (compare with the list on page 31).

Overall, erCOSA successfully implements the entity recognition task required by COSA. It proves that finite-state devices are a valid means to achieve this, even though they are slow at solving a subset of recognition problems. An implementation with better performance would have to use finite-state devices as well as other formalisms to share the task according to their respective strengths.

4 cosCOSA: Extracting entity relations

The final step in the overall COSA process is the generation of core sentences from the found entities. The implementation that was developed during this study is called cosCOSA and, again, depends on the results of the preceding step, in our case erCOSA. Core sentences relate a subject (a political actor) to an object (another actor or a topic) via a numerical predicate which represents the "direction" of the relationship.[54] These core sentences are the unit of choice to record information in the *NPW* project (Kriesi et al., 2006). Ideally, an article that has been processed by a COSAimplementation would generate only semantically correct core sentences. However, On the one hand, this would be a very challenging task linguistically. The preprocessing module (preCOSA for English articles) does supply syntactic information, which the extraction of relations could be based on. It is questionable whether a high percentage of correct core sentences could thus be achieved. Even if that was the case, COSA is not designed to do fully automated annotation and the resulting time savings for the person who checks the annotated articles would not be decisive. Still, COSA should at the very least suggest possibly correct core sentences. cosCOSA therefore ensures that, based on the entities found by erCOSA, it offers all potentially valid core sentences for manual review.

Since we are neither trying to use syntactic information to figure out whether a given entity should be the subject or the object of a core sentence, nor drawing on additional (lexical or semantic) information to determine the predicate value, our task is rather straight-forward. For all sentences that contain at least one actor and at least one additional entity, we have to build all subject-object pairs with an actor in the subject position. The core sentences will either be of the *AA* (*actor-actor*) or *AT* (*actor-topic*) type. If we look at the example sentence **(63)**, with **(64)** representing the relevant XML as output by erCOSA, we will

[54] While not presently implemented by COSA at all, even in manual annotation, the predicate only has the values *neutral* (0), *positive* (+1), or *negative* (-1).

see that the rule we have just formulated has to be refined to avoid generating core sentences that are predictably incorrect.

(63) *"Campbell's speech was too patriotic", said one journalist after Alastair Campell's address.*

(64)
```
<sentence id="X">
  <text>
    <token id="X-1" ... >‘‘</token>
    <token id="X-2" ... >Campbell</token>
    <token id="X-3" ... >’s</token>
    <token id="X-5" ... >speech</token>
    ...
  </text>
  <syntax> ... </syntax>
  <entities>
    <actor id="act-6009-50083_2" listid="act-6009-50083">
      <tokenref ref="X-2"/>
    </actor>
    <actor id="act-6009-50085_2" listid="act-6009-50085">
      <tokenref ref="X-2"/>
    </actor>
    <topic id="top-6-5000601_8" listid="top-6-5000601">
      <tokenref ref="X-8"
    </topic>
    <actor id="act-6009-50083_15_16" listid="act-6009-50083">
      <tokenref ref="X-15"/>
      <tokenref ref="X-16"/>
    </actor>
  </entities>
</sentence>
```

The second token ("Campbell") has been recognized as either Alastair Campbell (*act-6009-50083*) or Thomas Campbell (*act-6009-50085*).[55] The topic with the ID *top-6-5000601* has been assigned to "patriotic", and finally, tokens 15 and 16 again got annotated with *act-6009-50083*. Note that what we used to call "entity ID" is now the value of *listid*. The recognized entities also get a unique ID which has the relevant token numbers appended. If we went by the rule as defined above, the following subject-object pairs would be generated:

1. AA: act-6009-50083_2, act-6009-50085_2
2. AT: act-6009-50083_2, top-6-5000601_8

[55]This is because of ERCOSA's lack of forward resolution for ambiguous last names.

3. AA: act-6009-50083_2, act-6009-50083_15_16
4. AA: act-6009-50085_2, act-6009-50083_2
5. AT: act-6009-50085_2, top-6-5000601_8
6. AA: act-6009-50085_2, act-6009-50083_15_16
7. AA: act-6009-50083_15_16, act-6009-50083_2
8. AA: act-6009-50083_15_16, act-6009-50085_2
9. AT: act-6009-50083_15_16, top-6-5000601_8

Of these nine relations, however, four do not form valid core sentences. In 1 and 4, the subject and the object are assigned to the same token, which cannot logically represent two actors at the same time. In 3 and 7, the subject and the object are one and the same actor, which does not make sense semantically. We thus have to extend the rule: If the subject and the object share tokens, or if the subject and the object represent the same entity, the core sentence is invalid.

cosCOSA first parses the input XML and extracts all entities. Then, in keeping with the rules we have just set out, it would insert the following core sentences into our example XML:

(65)
```
<cores>
  <at_core>
    <subjectRef ref="act-6009-50083_2"/>
    <objectRef ref="top-6-5000601_8"/>
    <predicate>0</predicate>
  </at_core>
  <at_core>
    <subjectRef ref="act-6009-50085_2"/>
    <objectRef ref="top-6-5000601_8"/>
    <predicate>0</predicate>
  </at_core>
  <aa_core>
    <subjectRef ref="act-6009-50083_2"/>
    <objectRef ref="act-6009-50083_15_16"/>
    <predicate>0</predicate>
  </at_core>
  <aa_core>
    <subjectRef ref="act-6009-50083_15_16"/>
    <objectRef ref="act-6009-50085_2"/>
    <predicate>0</predicate>
  </at_core>
  <at_core>
    <subjectRef ref="act-6009-50083_15_16"/>
```

```
      <objectRef ref="top-6-5000601_8"/>
      <predicate>0</predicate>
    </at_core>
  </cores>
```

The zero value for predicates means "neutral". All of the tag and attribute names as well as the structure comply with with COSA specifications. cosCOSA also removes all syntactic annotation as well as the POS tag of all tokens to meet the predefined format. DOM-style XML manipulation is used for all these tasks. The output of cosCOSA is also the final output of the overall system that was developed for this thesis.

From a technical point of view, cosCOSA is a fairly unproblematic program which does not need to process large amounts of data and has none of the issues we have seen with erCOSA. Processing an article usually takes less than a seconds, so speed is not a concern, either. In terms of usability, cosCOSA inherits much from erCOSA. Its configuration file is very similar and it provides the same batch processing and logging facilities. More technical information can be found in the README, INSTALL and configuration file (section A.3, section C.3 and section B.2, respectively).

cosCOSA, in its current form, is a minimal implementation of COSA's relation extraction module. In building subject-object pairs, it uses basic rules to avoid logically or semantically invalid core sentences. There is no attempt to be intelligent about the allocation of the subject and object roles, and neither is the "direction" of the subject-object relationship evaluated. The present implementation focuses on generating all potentially correct core sentences, on outputting COSA-conform XML, and on usability. Developing an improved relation extraction module for COSA certainly is an interesting challenge for the future.

5 Conclusion

COSA is an attempt to automate the annotation of predefined political actors and topics as well as their interrelations in newspaper articles, using methods of natural language processing. Within that project, the present study implemented three modules: preCOSA for the tagging and parsing of English articles, erCOSA for the recognition of entities with the help of finite-state devices, and cosCOSA for relation extraction in the form of *core sentences*, which is the representation chosen for the *NPW* project, COSA's primary area of application.

Most of the effort was concentrated on the first two modules. preCOSA uses a pipeline which integrates a variety of tools, among them TreeTagger and Pro3Gres, to prepare English newspaper articles for further processing. erCOSA is the centrepiece of this thesis and explores the use of finite-state devices for the purpose of recognizing the relevant political entities, which are defined in a list, using a COSA-specified format. This was achieved by a scheme which represents articles as automata and entities as transducers. The result of the composition of an article FSA with all entity FSTs contains all necessary information to arrive at a fully annotated article. erCOSA successfully demonstrates the viability of an implementation of the entity recognition task that, for the actual recognition process, relies purely on the finite-state formalism. It has also become evident, however, that some problems, especially the resolution of name ambiguities, introduce a complexity which threatens the limit of what can sensibly be done with finite-state devices. A more efficient implementation should therefore combine the erCOSA's approach with other methods. cosCOSA, finally, fulfils but the minimal requirements. It extracts all entity relations (within a sentence) that could possibly make sense and expresses them as core sentences with a neutral predicate. Sadly, the syntactic information that is discovered by preCOSA remains untapped. As far as relation extraction is concerned, this is where the greatest potential for improvement lies. With regard to the first two modules, COSA could most benefit from even a basic form of pronominal

anaphora resolution, since the core sentence approach requires related entities to occur in the same sentence.

In this thesis, the first priority was not to probe the limits of what is possible with the latest and greatest methods of computational linguistics. First and foremost, the resulting system was required to be fit for productive application within the *NPW* project, which is carried out at the Department of Comparative Politics in Zurich in collaboration with other universities. Using a test corpus of nearly 800 articles, considerable effort thus went into ensuring good usability and robust operation. Ultimately, the fate of COSA as a whole will decide whether it will ever be possible to evaluate the qualitative and quantitative performance of the this implementation under "real-world" conditions, thus enabling future improvements.

References

Aït-Mokhtar, Salah and J. P. Chanod. 1997. Incremental finite-state parsing. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, pages 72–9, Washington DC, USA.

Carstensen, Kai-Uwe, C. Ebert, C. Endriss, H. Langer, S. Jekat, and R. Klabunde, editors. 2004. *Computerlinguistik und Sprachtechnologie*. Spektrum Verlag, Heidelberg.

Karttunen, Lauri. 1995. The replace operator. In *Proceedings of the 33rd conference on Association for Computational Linguistics*, pages 16–23, Association for Computational Linguistics, Morristown, USA.

Kleinnijenhuis, Jan, J. A. de Ridder, and E. M. Rietberg. 1997. Reasoning in economic discourse: An application of the network approach to the dutch press. In C. W. Roberts, editor, *Text analysis for the Social Sciences: Methods for Drawing Statstical Inferences from Texts and Transcripts*. Lawrence Erlbaum Associates, Mahwah, USA, chapter 11, pages 191–209.

Koskenniemi, Kimmo. 1997. Representations and finite-state components in natural language. In E. Roche and Y. Schabes, editors, *Finite-State Language Processing*. MIT Press, Cambridge, USA.

Kriesi, Hanspeter, E. Grande, R. Lachat, M. Dolezal, S. Bornschier, and T. Frey. 2006. Globalization and the transformation of the national political space: Six European countries compared. *European Journal of Political Research*, 45(6):921–56.

Mikheev, Andrei and S. Finch. 1997. A workbench for finding structure in texts. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, pages 372–379, Washington DC, USA.

Minnen, Guido, J. Carroll, and D. Pearce. 2001. Applied morphological processing of English. *Natural Language Engineering*, 7(3):207–23.

Mitkov, Ruslan, editor. 2003. *The Oxford Handbook of Computational Linguistics*. Oxford University Press, New York, USA.

Neumann, Günter and J. Piskorski. 2002. A shallow text processing core engine. *Computational Intelligence*, 18(3):451–76.

Ratnaparkhi, Adwait. 1996. A maximum entropy part-of-speech tagger. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 133–142, Philadelphia, USA.

Reynar, Jeffrey C. and A. Ratnaparkhi. 1997. A maximum entropy approach to identifying sentence boundaries. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, pages 16–19, Washington DC, USA.

Rinaldi, Fabio, G. Schneider, K. Kaljurand, M. Hess, C. Andronis, O. Konstandi, and A. Persidis. 2007. Mining of relations between proteins over biomedical scientific literature using a deep-linguistic approach. *Artificial Intelligence In Medicine*, 39(2):127–36.

Rinaldi, Fabio, G. Schneider, K. Kaljurand, M. Hess, and M. Romacker. 2006. An environment for relation mining over richly annotated corpora: the case of GENIA. *BMC Bioinformatics*, 7(Suppl 3):S3.

Roche, Emmanuel and Y. Schabes. 1997. Introduction. In *Finite-State Language Processing*. MIT Press, Cambridge, USA.

Schmid, Helmut. 1994. Probabilistic part-of-speech tagging using decision trees. In *Proceedings of International Conference on New Methods in Language Processing (NeMLaP)*, Manchester, UK.

Schmid, Helmut. 2005. A programming language for finite state transducers. In *Proceedings of the 5th International Workshop on Finite State Methods in Natural Language Processing (FSMNLP)*, Helsinki, Finland.

Schmid, Helmut, A. Fitschen, and U. Heid. 2004. SMOR: A German computational morphology covering derivation, composition and inflection. In *Proceedings of the 4th International Conference on Language Resources and Evaluation (LREC)*, pages 1263–6, Lisbon, Portugal.

Schneider, Gerold. 2004. Combining shallow and deep processing for a robust, fast, deep-linguistic dependency parser. In *European Summer School in Logic, Language and INformation (ESSLLI)*, Nancy, France.

Schneider, Gerold, F. Rinaldi, and J. Dowdall. 2004. Fast, deep-linguistic statistical dependency parsing. In *Proceedings of the Workshop on Recent Advances in Dependency Grammar, COLING*, pages 41–8, Geneva, Switzerland.

Schneider, Gerold, F. Rinaldi, K. Kaljurand, and M. Hess. 2004. Steps towards a GENIA dependency treebank. In *Third Workshop on Treebanks and Linguistic Theories (TLT-2004)*, pages 137–48, Tübingen, Germany.

Wüest, Bruno R. 2006. Pilotstudie zu einer Applikation für die computergestützte Kernsatzanalyse. Master's thesis, University of Zurich.

Wüest, Bruno R., A. Bünzli, and T. Frey. Unpublished. Semi-automatic annotation: Mining of relations between political actors and issues in newspaper articles. Unpublished, University of Zurich.

A Appendix: REAMDEs

A.1 preCOSA

```
# preCOSA Readme
# Author: Lukas Rieder
# Date: January 2008

** SYNOPSIS **

erCOSA is based on the NLP pipeline developed by Kaarel Kaljurand at the
University of Zurich for use with Gerold Schneider's parser Pro3Gres. The
pipeline has been slimmed down to what is needed for the COSA project and
extended with custom scripts to accommodate COSA-specific requirements.
Given a newspaper article in the COSA-defined XML format, it subjects that
article to tokenization, POS-tagging, lemmatization and syntactic
dependency parsing and adds the corresponding tags and attributes.

** FILES **

build.xml
    The buildfile for Apache Ant is the core of the pipeline.

components/
    Contains auxiliary scripts and various files for each stage of
    the pipeline.

INSTALL
    Installation notes.

nlppl.log
    Logs pipeline output when using run.sh.

nlppl.properties
    Configuration file for the pipeline.

nlpplrun/
    All intermediate results of processing a file through the pipeline
```

```
    are stored here.

pipline.txt
    Sketches the individual stages of the preCOSA pipeline.

README
    This file.

run.sh
    A wrapper for the pipeline which allows batch processing (see below).


** USAGE **

It is possible to manually invoke Apache Ant for processing an article. The
target for the end result is "cosa":

$ ant -Din <input_article> cosa

There is a wrapper script which facilitates batch processing a number of
articles. It takes all articles which match a certain filename pattern (see
inside run.sh), runs them through the pipeline and drops them back into the
same directory with a different filename (again, see run.sh for
customization). The pipeline's status output is captured in nlppl.log and
batch processing terminates on any error.

$ ./run.sh <dir_of_input_articles>

** TWEAKING **

There is room for improvement (and potential for bugs) at every stage of
the pipeline. It is beyond the scope of this document to describe every
decisions that was taken and all the scripts that do their work along the
way. In general, the scripts in components/ should contain some form of
documentation and together with build.xml, one should be able to make some
sort of sense of the whole process.
```

A.2 erCOSA

```
# erCOSA Readme
# Author: Lukas Rieder
# Date: January 2008

** CONTENTS **
```

Synopsis
Files
Usage
Known Issues

** SYNOPSIS **

Given an XML article as produced by preCOSA, erCOSA will find and annotate (named) entities as defined by a list in the format agreed upon within the COSA project. It will do so with the help of finite state transducers (FSTs), using the open-source tool SFST via its Python bindings, pysfst.

Both the article as well as all entity "recipes" are transformed into the description of an automaton/transducer ("article FSTs" and "entity FSTs"). Then the each of the article FSTs is "composed" (one of the possible operations on transducers) with all entity FSTs, which after some post-processing yields the entities that were found in the given article. They are, in the COSA-specified manner, added to the XML which is written out under a new name.

It should be noted that using FSTs for the job of recognising named entities is not exactly a well-treaded path. It is upon the task of building transducers to recognise relevant entities that the quality and efficiency of erCOSA hinges. The current implementation is a decent compromise between the two, but it is yet at a "high" (or indeed "low", depending on your metaphor) level. The transducers deliver what they promise in functionality, but could certainly do better and most definitely do it a lot faster. See also "Known Issues" for some of the limitations.

** FILES **

article.py
config.py
entities.py
process.py
shared.py
templates.py
update.py
    The python modules which make up the actual program.

INSTALL
    Installation instructions.

```
README
    This file.

uk_2005.config
    A sample configuration file (specific rather than exemplary, in fact...).

lib/
    "Global" FST-PL files which are compiled online.
    anyfirst.fst
        Describes a token which is (could be) a first name.
    chars.fst
        Lists all characters that may appear in "free" text.
    clean.fst
        Removes all tokens not tagged with an entity from a transducer
        representing an article.
    common.fst
        Defines common variables for all FSTs and is included in all of them.
    nofirst.fst
        Defines a token that is not (shouldn't be) a first name.
    numbers.fst
        Defines symbols for sentence and token numbers.
    ptts.fst
        Defines symbols for all POS tags of the Penn Treebank Tag Set.
    reduce.fst
        Chops up an article and returns only unique tagged tokens in an
        easy-to-parse format.

templates/
    A number of templates for often-used building blocks of entity FSTs.


** USAGE **

First of all, you need to make sure you have an appropriate config file for
the task at hand - take your cue from uk_2005.config, it has explanations
for all the options. Basically, you only have to set the paths to your
articles and the location of a log file, should you wish to use one.

1) Updating entities

As I said above, there are two distinct tasks to erCOSA. To make sure all
the transducers which are responsible for recognising your entities are in
sync with the provided list(s) and up to date with any configuration
options, you should always run the update task before processing articles:
```

```
$ ./update.py [--debug] <config>
```

Specifying "--debug" will lead to more messages. "<config>" is your config file, e.g. "uk_2005.config".

This will parse your entity list(s) and generate a transducer description (in the language SFST-PL - these are the "*.fst" files). If there are new or changed entities, the relevant files will be created or updated. Depending on your configuration, files for entities which are no longer on the list will be deleted. In the next step, all entity FSTs which are new or have been changed will be (re)compiled. If a file in the "lib/" directory or a config option that applies to all transducers has been changed, all FSTs will be compiled. This will probably take 'hours'.

2) Processing articles

Once your entities are ready, you can move on the actual processing of your articles:

```
$ ./process.py [--debug] <config>
```

The arguments are the same as above.

Note that input articles (as per the config file) will only be processed if they are newer than the corresponding output file. You might want to delete all output articles if you have changed your entity list(s).

3) Hint: parallel processing

If you have multiple CPUs (or CPU cores) at your disposal, you can multiply the speed of erCOSA, because it does not inherently distribute its load very much at all. erCOSA has a mechanism to lock files for other erCOSA processes, so running multiple instances in parallel poses no danger to the integrity of its data. To max out 4 CPU cores, you'd have to run at least 4 instances of erCOSA. I suggest you make use of "nice" in order not to bog down the entire system. Note that erCOSA log messages include a process ID so you can discern the processes running in parallel.

4) Hint: silence SFST

Because SFST continuously outputs ominous numbers (when minimising transducers, I think) which pysfst fails to suppress and erCOSA can do nothing about, it is a good idea to redirect erCOSA's standard output to /dev/null because the constant need to write to the console causes

considerable but completely unnecessary system load. All of erCOSA's log messages are written to StdErr, so for normal usage, you should not miss anything.

** KNOWN ISSUES **

1) Memory leak

There are inherent issues with memory management when using SWIG, as pysfst does, to integrate, in this case, C++ code with Python. While I could not track down the exact cause, the problem is related to the SWIG proxy classes not offering a __del__ method, hence Python's garbage collector only trashes the pointer to the C++ object rather than the actual object. One of the supplied patches has a naive and probably incorrect fix for at least the Transducer class, but memory usage still rises significantly over time. The initial memory footprint is around 100 MB initially, but can grow to 10x the amount after many hours of continuous processing.

2) Segmentation faults with big transducers

SFST (and this does seem to be a problem of the original program rather than the Python bindings) seems to have issues with very large transducers, as can be necessary to represent long articles. There are several workarounds in place, the least convenient of which is this: If the transducer for representing an article gets too big - the critical size is merely the result of observing crashes and might depend on many factors - the remaining sentences are simply ignored. At least a (major) part of the article can thus be processed.

3) Alphabet definition

Transducers have to rely on a well-defined alphabet of symbols which might possibly occur anywhere in the strings it processes. While erCOSA being unicode compatible, including all unicode characters in the alphabet would make processing unbearably slow. It is likely, then, that erCOSA will at some point encounter an unknown character and fail, which should trigger an error message hinting at the cause. You then have to find the offending character and include it in "chars.fst". Unfortunately, this will entail the recompilation of all entity transducers.

4) Speed

Compiling an entity FST usually takes a few minutes (names take longer than keywords), while very complex cases (many name variations) can take more than an hour. (Times taken using one core of an AMD Opteron 285).

5) No forward reference resolution for names

As the only linguistic shortcoming (in relation to what I or the COSA project deemed necessary and feasible), ambiguous last names which occur prior to any first name that goes with it (as is often the case in titles), there is no disambiguation. This was omitted because of the anticipated speed penalty, which, because of the potentially infinite scope of the rule should be considerable. There might, however, be fast and clever ways around this which were not investigated.

6) Uncontrolled output from SFST

pysfst does a pretty good job at suppressing most of SFST's output, but there are leftovers. You can see rows of characters ('#' mostly, but I've seen others) when it reads in transducers and some sort of continuous count when it is minimising transducers. erCOSA can do nothing about this - see the hint above to silence this.

A.3 cosCOSA

```
# cosCOSA Readme
# Author: Lukas Rieder
# Date: January 2008

** CONTENTS **

Synopsis
Files
Usage
Known Issues


** SYNOPSIS **

Given an XML article as produced by erCOSA, cosCOSA will, for each
sentence, look at the found entities and combine them into "core sentences"
as appropriate. The current approach is to produce all "core sentences"
that could potentially be relevant; each actor is combined with each other
actor into an AA core unless the other actor has the same entity ID (i.e.
it is really the same actor) or the other actor shares tokens with the
first actor (i.e. there is an overlap and the two actors are really
alternative matches.) Similarly, each actor is combined with each topic to
form an AT core. The resulting core sentences are inserted into the input
```

XML in the agreed COSA format and the XML is written to an output file.

** FILES **

config.py
coscosa.py
shared.py
    The python modules which make up the actual program.

INSTALL
    Installation instructions.

README
    This file.

uk_2005.config
    A sample configuration file (specific rather than exemplary, in fact...).

** USAGE **

First of all, you need to make sure you have an appropriate config file for the task at hand - take your cue from uk_2005.config, it has explanations for all the options. Basically, you only have to set the paths to your articles and the location of a log file, should you wish to use one.

To start processing articles, do this:

$ ./coscosa.py [--debug] <config>

Specifying "--debug" will lead to more messages. "<config>" is your config file, e.g. "uk_2005.config".

Input articles (as per the config file) will only be processed if they are newer than the corresponding output file.

Please note that because cosCOSA is reasonably fast (no more than a few seconds per article), no file locking mechanism has been implemented. You must therefore not run more than one cosCOSA process accessing the same articles in parallel.

** KNOWN ISSUES **

cosCOSA should be quite robust since it does fairly straight-forward DOM-style XML manipulation. There are no known issues.

B Appendix: Sample Configurations

B.1 erCOSA

```
# ----------------------------------------------------------------------
# erCOSA configuration
#
# - adhere to python's ConfigParser conventions
# - use $TAB and $SPACE if you need to specify the according whitespace
#   on its own
# - where more than one value is expected, use commas (,) to separate them
# - %(erCOSADir)s is implicitly defined and always resolves to the path
#   containing the erCOSA python modules.
# ----------------------------------------------------------------------

[Main]

####################
## Basic options ##
####################

# Path to the survey-specific data
DataDir: %(erCOSADir)s/../data/uk_2005

# One or more entity list files
EntityLists: %(DataDir)s/lists/actors-1.txt,
             %(DataDir)s/lists/actors-2.txt,
             %(DataDir)s/lists/topics.txt

# FST file that defines all POS tags
POSTags: %(erCOSADir)s/lib/ptts.fst

# POS tags which indicate a proper name
NamePOSTags: NNP, NNPS

# Articles for processing will be read from this directory ...
InputArticleDir: %(DataDir)s/articles
```

```
# ... if they have this suffix.
InputArticleSuffix: .precosa.xml

# Processed articles will be written to this Directory ...
# (can be the same as InputArticleDir)
OutputArticleDir: %(InputArticleDir)s

# ... with this suffix.
OutputArticleSuffix: .ercosa.xml

# All messages will be logged to this file, leave empty to disable logging
LogFile: %(DataDir)s/ercosa.log


# ------------------------------------------------------------------------
# There should not normally be a need to change options below this line.
# ------------------------------------------------------------------------


######################
## Advanced options ##
######################

# Show SFST output during compilation
VerboseCompilation: no

# Remove all entity FSTs which are not on the current list(s)
PruneStaleEntities: yes

# A hash of all compilation includes will be stored here - used to detect
# changes
HashFile: %(DataDir)s/.includehash

# Entity FSTs will be placed in this directory
EntityDir: %(DataDir)s/entities


########################
## Entity List format ##
########################

# Field delimitor
FieldDelim: $TAB

# Name of first name field
```

```
FNField: forename

# Name of last name field
LNField: surname

# Name of keyword field
KWField: keyword

# Fields which can (and should) be ignored
IgnoreFields: name

# Keyword search: within same sentence, free order
AND: &

# Keyword search: only valid following AND
NOT: !

# Keyword search: concatenation, immediate neighbours in order
PLUS: $SPACE

# RegExp of a valid keyword search
KWSearch: [^&]+(\&\!?[^&]+)*

# RegExp of a valid entity ID
EntityID: [a-z0-9-]+

# Entity IDs starting like this indicate actors
ActorPrefix: act-

# Entity IDs starting like this indicate actors
TopicPrefix: top-


########################
## Article XML format ##
########################

# RegExp of a valid token ID
TokenID: [a-z0-9-_]+

# The following options allow for changes in the XML format
ArticleElement: article
SentenceElement: sentence
TokenElement: token
ActorElement: actor
```

```
TopicElement: topic
IDAttribute: id
LemmaAttribute: lemma
POSAttribute: POS
```

B.2 cosCOSA

```
# ----------------------------------------------------------------------
# cosCOSA configuration
#
# - adhere to python's ConfigParser conventions
# - %(cosCOSADir)s is implicitly defined and always resolves to the path
#   containing the cosCOSA python module.
# ----------------------------------------------------------------------

[Main]

####################
## Basic options ##
####################

# Path to the survey-specific data
DataDir: %(cosCOSADir)s/../data/uk_2005

# Articles for processing will be read from this directory ...
InputArticleDir: %(DataDir)s/articles

# ... if they have this suffix.
InputArticleSuffix: .ercosa.xml

# Processed articles will be written to this Directory ...
# (can be the same as InputArticleDir)
OutputArticleDir: %(InputArticleDir)s

# ... with this suffix.
OutputArticleSuffix: .cosa.xml

# All messages will be logged to this file, leave empty to disable logging
LogFile:


# ----------------------------------------------------------------------
# There should not normally be a need to change options below this line.
# ----------------------------------------------------------------------
```

```
########################
## Article XML format ##
########################

SentenceElement: sentence
EntitiesElement: entities
SyntaxElement: syntax
TokenElement: token
TokenRefElement: tokenref
CoresElement: cores
ActorElement: actor
TopicElement: topic
AACoreElement: aa_core
ATCoreElement: at_core
SubjRefElement: subjectRef
ObjRefElement: objectRef
TopicRefElement: topicRef
PredicateElement: predicate
RefAttribute: ref
POSAttribute: POS
InfoAttribute: info
```

C Appendix: INSTALLs

C.1 preCOSA

```
# preCOSA Installation Notes
# Author: Lukas Rieder
# Date: January 2008

preCOSA is a customized version of NLPPL as developed by Kaarel (see
README). Becaue NLPPL relies on a variety of software components, some of
which are distributed in binary form only (LTCHUNK) or not released yet
(Pro3Gres), it is not, for the time being, very portable at all.

There are plans to improve this situation, but at the moment, setting up
new installations is not encouraged.
```

C.2 erCOSA

```
# erCOSA Installation Notes
# Author: Lukas Rieder
# Date: January 2008

** REQUIREMENTS **

erCOSA itself is written in Python and requires at least version 2.3. The
only external module needed is pysfst (>= 1.1.2), the Python bindings for
SFST (>= 1.1). My own code should be platform independent but SFST is best
supported on Linux and will certainly not run on Windows. In case you
wonder, SFST will run on both 32-bit and 64-bit systems.

** STEP-BY-STEP **

1) Before you start

The installation of pysfst is neither foolproof nor very difficult. What
makes it a little bit daunting is having to apply additional patches on top
```

of pysfst's own multi-step instructions. pysfst needs SFST sources of the correct version, but you do not need to install SFST separately, unless you want to use its native, stand-alone binaries.

2) You need

- SFST (by Helmut Schmid), tested with version 1.1
  http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html

- pysfst (by Toni Arnold), tested with version 1.1.2
  http://home.gna.org/pysfst/

- flex, tested with version 2.5.33

- bison, tested with version 2.0, 2.3

- swig, tested with version 1.3.31 (optional, see below)

You will probably have the last three installed already, but the versions can be a problem (at least that's what I found with an old version of flex).

3) Installing pysfst

- Read the included README.

- After step 2: apply fix_fclose_bug.patch to SFST/src/interface.C

- Step 3: as you can see, rebuilding the SWIG files is optional. Unless you don't mind the additional dependencies or require the functionality (erCOSA doesn't), I suggest you remove the -DGRAPHWIZ and -DWXWINDOWS keywords as indicated and do the rebuild. Also note that my less-than-optimal patch below might complain if you don't because your sfst.py will differ.

- After step 4, you'll have a subdir called "build". Copy the two files below "lib-<arch>" (sfst.py, _sfst.so) into Python's system-wide site-packages or make sure they're in your PYTHONPATH. (See Python docs if you don't know what this is about.)

- Apply fix_memory_leak.patch to sfst.py.

4) erCOSA itself is ready-to-use, no installation required.

C.3 cosCOSA

```
# cosCOSA Installation Notes
# Author: Lukas Rieder
# Date: January 2008

cosCOSA consists of three Python modules and does not have any extra
requirements. It is also platform independent. I think Python >= 2.3 should
be fine.
```

Curriculum vitae

Personalien

Name	Lukas Rieder
Adresse	Untere Halde 15, 5400 Baden
Email	lukas.rieder@email.ch
Geburtsdatum	08. August 1979
Heimatort	Gränichen AG
Zivilstand	ledig

Ausbildung

1996–2000	Kantonsschule Baden
	Typus E: Wirtschaft
seit 2000	Universität Zürich
	Englische Sprach- und Literaturwissenschaft
	Informatik (Vertiefungsgebiet: Computerlinguistik)
	Religionswissenschaft
2003–2004	University of Sheffield
	Auslandstudium (ERASMUS), zwei Semester

Anstellungen

bis 2001	Diverse Ferienanstellungen
2001–2002	Technical Supporter
	Digicomp AG, Schlieren
2002–2005	Technical Supporter, stellvertretender Teamleader
	Tempobrain AG, im Einsatz bei Cablecom AG, Otelfingen
seit 2005	IT Systems Engineer
	Digicomp Academy AG, Zürich