Saxon: An Extensible Multimedia Annotator

Mark A. Greenwood, José Iria, Fabio Ciravegna

Department of Computer Science
University of Sheffield
Sheffield, S1 4DP, UK
{m.greenwood, j.iria, f.ciravegna}@dcs.shef.ac.uk

Abstract

This paper introduces Saxon, a rule based document annotator that is capable of processing and annotating several document formats and media, both within and across documents. Furthermore, Saxon is readily extensible to support other input formats due to both it's flexible rule formalism and the modular plugin architecture of the Runes framework upon which it is built. In this paper we introduce the Saxon rule formalism through examples aimed at highlighting it's power and flexibility.

1. Introduction

The application of natural language processing (NLP) to real-world data and tasks is complex, and has become significantly more so since web and multimedia data became commonplace. Novel, more complex tasks have emerged that: 1) work on different kinds of text (e.g. web people disambiguation (Artiles et al., 2007), sentiment analysis (Strapparava and Mihalcea, 2007)), 2) require taking into account not only linguistic features but also the underlying structure of the data (see, for instance, Minkov et al. (2006) and Zhu et al. (2007)), and 3) require handling large volumes (Popov et al., 2004a) and heterogeneity (Arasu and Garcia-Molina, 2003) of documents coming from third party controlled sources.

One of the issues encountered, as the amount of available data increases, is the choice of algorithms to extract useful information. Very large datasets exclude the possibility of exhaustive annotation. This has led to a rise in the number of unsupervised and semi-supervised methods being reported in the literature. Other related work includes numerous methods that employ manually developed rules to a wide range of natural language processing problems. Such methods, hereafter referred to as rule based methods, rely on experts of the problem domain for the manual elicitation of declarative rules that are capable of capturing patterns in the textual data and to take some action on the matched text fragments. Despite the costs associated with manually creating and maintaining these rules, such methods still constitute the best or, sometimes, only viable solution in many applications, where they replace or complement other methods, most notably machine learning-based methods that are capable of learning from examples marked in the text by non-expert users. Recent examples of application domains where systems that employ rule based methods have been used include market monitoring and technology watch (Maynard et al., 2005), semantic annotation of documents (Reeve and Han, 2005; Popov et al., 2004b), query answering (Buitelaar et al., 2006) and biomedical text mining (Hirschman et al., 2005).

The application of rule based systems to new application domains presents, generally, two main challenges: making sure the rules are expressive enough to capture the patterns of interest in the text, and coping with new document formats and media. These two aspects are intimately related, as the expressiveness power of rules depends not only on the grammar defined to enable their declarative specification, but also on the information which the system is capable of accessing from the underlying textual data. For instance, in order to capture text fragments that are marked as *italic* in a OpenDocument file, both grammar and access mechanism must support such style elements. A more complex example would concern rules that work across media, e.g. capturing text fragments surrounding images depicting cars in the corpus.

As application domains become more and more demanding, recent interest has moved to handling documents containing different formats and media types, e.g. plain text, HTML, and OpenDocument. Unfortunately, the majority of the current systems are not able to represent or make use of the information in these different format and media types. To address this issue we have designed and developed Saxon, an extensible rule based system built upon the Runes framework. Saxon combines the following key features:

- handles multimedia data (via the underlying framework);
- matches patterns across documents and media;
- features an expressive rule formalism;
- it is readily extensible to support new input formats and media.

Due to its expressive rule formalism, Saxon can be used to perform several tasks including automated document processing (conversion, segmentation, etc.) and classification tasks such as text categorization and entity and relation extraction. Conversely, due to its multimedia handling and extensibility features, Saxon can be easily ported to new application domains.

The rest of this paper describes the underlying Runes framework and Saxon's data representation capabilities and rule formalism. It also provides examples which explain how both Runes and Saxon work and their practical applications.

2. The Runes Framework

In this section we present an open-source framework, called Runes[1] (previous work on this framework was presented by Iria and Ciravegna (2006)), which handles, on behalf of the designer/implementer, several important aspects related to the representation of language resources: it automates the selection of an optimal underlying data structure for holding the data at any time during processing; it provides support for expressive data models up to the level of hypergraphs; it enhances portability by featuring a plugin framework and encouraging developers to think in terms of small modular processing units; it integrates data by providing unique identifiers to stored data and merging those that are identical; it orchestrates execution of external tools by running a dependencies resolution algorithm that determines which should run and in which order; and it supports processing several data formats and media thanks to the numerous plugins that accompany the framework.
The rest of this section describes the frameworks data representation formalism and plugin execution strategy.

2.1. Representation Formalism

From a mathematical standpoint, the data representation expressiveness offered by Runes is that of a hypergraph. From a framework perspective, this is achieved via the concepts of *stone* and *runestone*.

Definition 1. A pluggable data structure $\mathcal{S}$, or simply *stone*, is a 3-tuple $(\mathcal{D}, i, c)$. $\mathcal{D} \subseteq \mathbb{N}_1 \times \ldots \times \mathbb{N}_k$ is a set of data *instances*, where k is the *arity* of the instance. The injective functions $i : \mathcal{D} \to \mathbb{N}$ and $c : \mathcal{D} \to \mathcal{T}$ associate an *unique identifier* and a *content object* to an instance, respectively. In case $\mathcal{T} \equiv \mathbb{R}$, the content may be interpreted as a (real-valued) *weight*. A 1-ary instance constitutes a *node* in the hypergraph. A k-ary instance, $k \geq 2$, takes k unique identifiers from $1 \leq m \leq k$ stones, thereby forming an edge in the hypergraph.

Definition 2. A common data representation, or simply *runestone*, is a map $l \mapsto \mathcal{S}$, where $l \in \mathcal{L}$ is a *semantic type label*.

In NLP applications, a typical runestone consists of data structures for the semantic types $document$, $sentence$, $token$, $part\_of\_speech$, $sentence\_in\_document$ and $token\_has\_part\_of\_speech$, amongst others. A $part\_of\_speech$ stone $\mathcal{S}_{pos}$ is typically a set of 1-ary instances (nodes), which contain string content values such as 'NNP' and 'DT' and are uniquely identified by mapping those strings to integers. A $token\_has\_part\_of\_speech$ stone is typically a set of 2-ary instances (edges) whose first component is the identifier of an instance $d_1 \in \mathcal{S}_{token}$, and the second component the identifier of an instance $d_2 \in \mathcal{S}_{pos}$.
Note that the framework's ability to integrate data is given by function i and the fact that $\mathcal{D}$ is a set. For example, all insertions of a node with content 'NNP' into $\mathcal{S}_{pos}$ map to one single node, and generate the same identifier for use by other stones.

[1]`http://sourceforge.net/projects/runes`

2.2. Plugin Execution Strategy

From an architectural standpoint, Runes implements the blackboard architectural pattern (Buschmann et al., 1996). Processing is done by pluggable modules, which communicate and integrate data via a runestone. In Runes, pluggable processing modules $r_{1..n}$ read and write from a common data structure b, composed of structures holding data for semantic types $s_{1..m}$. Modules whose input and output lie in b (e.g., r_2) are automatically created by the framework by analysing the requirements imposed by the inputs and outputs that lie outside b (e.g., input of r_1 and output of r_3). In NLP applications, this means the application designer does not have to manage the intermediate steps of the pipeline.

Definition 3. A pluggable processing module $\mathcal{R}$, or simply *rune*, is a 3-tuple $(\mathcal{I}, \mathcal{N}, \mathcal{P})$. $\mathcal{I}$ denotes a set of instructions that modify the runestone. The set of *required labels* $\mathcal{N} \subseteq \mathcal{L}$ indicates which stones need to be available to the rune prior to execution. The set of *provided labels* $\mathcal{P} \subseteq \mathcal{L}$ indicates which stones will be created or modified by the rune when executed.

For example, a rune wrapping an existing part-of-speech tagger would require, prior to execution, the presence in the runestone of an edge between each document node and its first token node, and of edges to gather adjacent tokens, that is, $\mathcal{N}_{pos} = \{token, document\_has\_first\_token, next\_token\}$. The same rune would attach part of speech tags to each token, that is, $\mathcal{P}_{pos} = \{part\_of\_speech, token\_has\_part\_of\_speech\}$.
The framework encourages a modular design of the collection of runes for portability and reuse. The processing model aims at several runes to be composed for execution, such that a rune's required set is fulfilled by one or more other runes' provided sets. When $\mathcal{P}_i \subseteq \mathcal{N}_j$, we say rune j depends on rune i. Orchestration concerns are hidden away from the user by automatically resolving such dependencies.
The dependencies resolution algorithm in Figure 1 finds a minimal set of runes that can be composed to execute a given processing task, defined in terms of its sets of required and provided semantic type labels. Having found this minimal set, an execution schedule can be determined and a data model can be inferred. The schedule groups runes into execution stages, where runes in the same stage have no dependencies amongst them and may thus be run in parallel. The data model $\mathcal{M} \subseteq \mathcal{L}$ specifies which stones will be created and modified during processing (one per label in $\mathcal{M}$). That is, when running a scheduled rune r, instances of a semantic type with label l such that $l \notin \mathcal{M} \cap \mathcal{P}_r$ are filtered out.
By handling all aspects of orchestration (which runes to run when, and which data model to subscribe to), Runes frees the application designer to think about the data processing task solely in terms of input and output type labels. In NLP applications, this means the intermediate steps of the pipeline do not have to be managed. For example, to collect all part-of-speech tags present in a corpus, the designer would specify as input the provided set $\mathcal{P} = \{doc\_url\}$ and as output the required set $\mathcal{N} = \{part\_of\_speech\}$,

```
resolve($N$, $P$, $R$, $S$)
1. $M \leftarrow N \setminus P$
2. if $M = \emptyset$ then record solution $S$
3. else foreach $r \in R$
4.  if $N_r \subseteq M$ then
5.   resolve($N \cup N_r$, $P \cup P_r$, $R \setminus \{r\}$, $S \cup \{r\}$)
6. rank solutions by $|S|$
```

Figure 1: The dependencies resolution algorithm. Given required (N) and provided (P) sets semantic type labels, the algorithm recursively selects runes from a rune set R, seeking to drive the number of missing labels M down to zero. Each of the solutions found is a runes set S satisfying the condition $N \cup (\bigcup N_r) = P \cup (\bigcup P_r), r \in S$.

from which the framework would infer a data model such as $\mathcal{M} = \{document,\ doc\_url,\ doc\_has\_first\_token,\ next\_token,\ token\_has\_part\_of\_speech,\ part\_of\_speech\}$ by automatically composing five runes[2]:

- $\mathcal{S}_u$=($\mathcal{I}_u$, $\emptyset$, $\{doc\_url\}$), where $\mathcal{I}_u$ inserts user-specified urls into the runestone;
- $\mathcal{S}_f$=($\mathcal{I}_f$, $\{doc\_url\}$, $\{document,\ doc\_has\_url,\ doc\_content,\ doc\_has\_content\}$), where $\mathcal{I}_f$ fetches document content from the urls and creates a document node that binds url and content;
- $\mathcal{S}_t$=($\mathcal{I}_t$, $\{doc\_content,\ doc\_has\_content\}$, $\{token,\ string,\ token\_has\_string,\ doc\_has\_first\_token,\ next\_token\}$), where $\mathcal{I}_t$ tokenises the document contents and generates a token sequence structure with attached token strings;
- $\mathcal{S}_p$=($\mathcal{I}_p$, $\{next\_token,\ doc\_has\_first\_token,\ token\_has\_string,\ string\}$, $\{part\_of\_speech,\ token\_has\_pos\}$), where $\mathcal{I}_p$ attaches part-of-speech tags to tokens;
- $\mathcal{S}_g$=($\mathcal{I}_g$, $\{part\_of\_speech\}$, $\emptyset$), where $\mathcal{I}_g$ collects part-of-speech tags from the runestone into a user-specified collection.

3. Handling Diverse Formats and Media

Saxon, builds on the framework introduced in the previous section, relying on the common data representation as a way efficiently integrate information coming from heterogeneous sources in an extensible and flexible way. This means that documents in the currently supported (plain text, HTML and OpenDocument) formats are integrated into one single representation of their original contents, which, crucially, allows for an uniform mechanism to access them. Saxon's rule formalism then builds upon this access mechanism to recognize complex patterns in the data, decoupling pattern recognition from data specific issues.
Throughout the rest of this paper we will use the simple example sentence "Dr John Smith wrote a paper." to illustrate the behaviour of Saxon. We do not need to specify the format of the document from which the sentence was taken as the Runes framework will abstract away from the original format to a common representation. Figure 2 shows a section of the hypergraph constructed by Runes to represent the example sentence. In this representation, each token is represented by a separate node in the graph, each of which are linked to the previous and following tokens and the sentence in which it appears. As well as representing the structure, the content of the document is represented (at both the sentence and token level) as well as the part-of-speech tags for each token. The edge labels are not shown in the diagram but they include (for the purpose of further discussion) the self explanatory $next\_token$, $previous\_token$, $token\_has\_string$ and $token\_has\_part\_of\_speech$. This is an intuitive representation of text that will be used in the examples throughout the remainder of this paper.

4. Rule Formalism

Saxon rules are capable of capturing patterns in multimedia data and to take some action on the matched media fragments. The former is typically called the left-hand side of the rule (LHS), whereas the later is known as the right-hand side (RHS). Saxon rules are very expressive because the LHS is based on the idea of graph walks and because the expressiveness of the RHS is only limited by the expressiveness of the Java language. This will be explained in what follows by way of the example rule in Figures 4 and 5, a rule designed to recognise the names of people which is typically part of any named entity recogniser.

4.1. Left-Hand Side Formalism

The LHS of each rule determines which portions of the document representation are selected and passed to the RHS. The LHS is a regular expression over edge types in the document representation, allowing for named edge traversal to collect nodes in the graph, and also for 'feature' checking at each node. Feature checking involves checking for the presence of edge types and content nodes that are not part of the actual path selected by the rule. This allows, for instance, for a rule to specify that an edge is only traversed if it leads to a node representing a token with a 'NNP' part-of-speech (POS) label. Of course nothing limits feature checking to a single feature or to directly connected nodes. For example, given the text representation of Figure 2 we could easily define a rule including a constraint that an edge is only traversed if it leads to a token whose POS label is 'NNP' and that the token is in a sentence which starts with the token 'The'. In other words, the complexity of the LHS of a rule is limited only be the complexity of the data encoded in the hypergraph over which the rule will operate.
The example rule in Figure 4 uses many of the capabilities of Saxon to develop a rule to find the names of people within documents. The two main sections of the LHS of the rule are illustrated in Figure 3.
The first part of the rule matches just those tokens which represent a given string. In this instance, tokens whose string matches the regular expression `Mr|Dr|Mrs|Miss` will be selected. This illustrates that nodes can be selected via the content using regular expressions prefixed by ~. To

[2]Note that for sake of brevity we only show a subset, relevant to the example, of the required and provided sets.

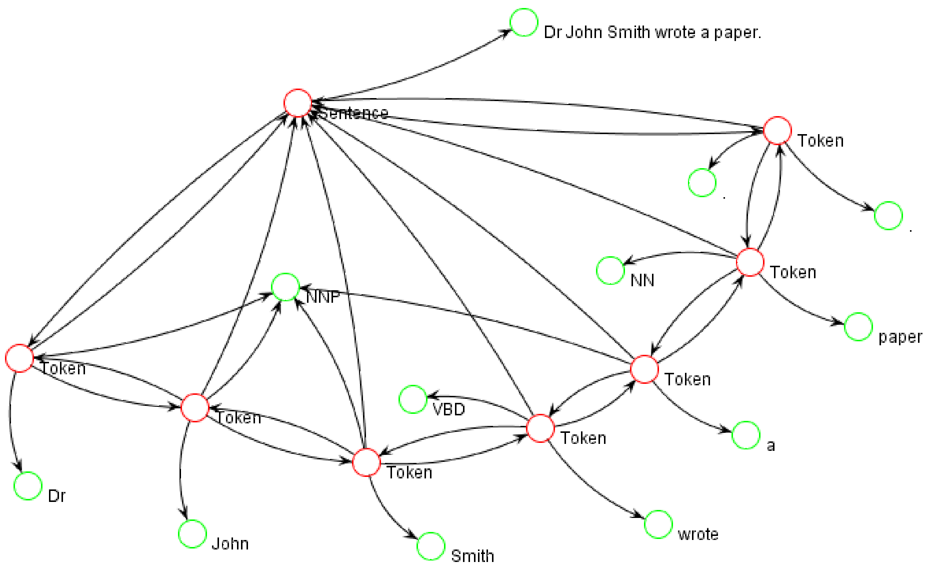

Figure 2: Example graph representation of the sentence ‘Dr John Smith wrote a paper.’.

1. `(next_token{token_has_string{~Mr|Dr|Mrs|Miss}})`
2. `(next_token{token_has_part_of_speech{=NNP}})+`

Figure 3: Breakdown of LHS from Example Rule

```
Rule:Person
(
   (next_token{token_has_string{~Mr|Dr|Mrs|Miss}})
   (next_token{token_has_part_of_speech{=NNP}})+
)
```

Figure 4: Basic Saxon rule for recognising the names of people.

```
Rule:Person
(
   (next_token{token_has_string{~Mr|Dr|Mrs|Miss}})
   (next_token{token_has_part_of_speech{=NNP}})+
)
=>
{
   try {
      java.net.URI edge = builder.getModel().getTypeFrom("token_string");
      int type = builder.getModel().getIndexOf(edge);
      for (Path path : paths) {
         for (RepresentationNode node : path.getNodes()) {
            System.out.print(node.follow(type)+" ");
         }
         System.out.println();

         addSaxonNode(builder,"Person",path);
      }
   }
   catch (Exception e) {
      e.printStackTrace()
   }
}
```

Figure 5: Saxon rule for recognising the names of people including a Java based RHS.

select based on string equality instead of using a regular expression the ~ is simply replaced by a =.
The use of = instead of ~ can be seen in the second part of the rule. The second part of the rule traverses the graph while the POS tag of the token being represented is 'NNP' (i.e. the token is a proper noun). As in any other regular expression based language the + means that the preceding section must match at least once for the rule as a whole to match.
These two sections are then used sequentially to build the complete LHS seen in the example rule in Figure 4. In essence the rule selects those subgraphs that represents sections of a document in which a known title token is followed by one or more proper nouns – a common rule in named entity recognisers for finding the names of people.

4.2. Right-Hand Side Formalism

One of the main strengths of Saxon is the flexible nature of the RHS of each rule, which is executed when the LHS matches against the document representation. There are currently three different possible forms that the RHS can take:

Left Blank If a rule does not specify a RHS (as in Figure 4) then, a standard annotation (i.e. new nodes are added to the underlying graph) is added to the document representation. The annotation records the rule name as the annotation type. For example, when the rule in Figure 4 matches an annotation of type Person is stored against the matching subgraph.

Simple Entity Type To change the default annotation type, the RHS can specify an alternative using the form `[Type]`.

Unrestricted Java Code The most flexible type of RHS allows for unrestricted Java code to be specified within the rule. This allows for any required actions to be performed upon a rule matching the document representation. A number of utility methods are available to help in RHS rule development, full details of which can be found in the Saxon documentation[3].

Figure 5 shows an expanded version of the rule in Figure 4. This expanded rule contains a RHS of unrestricted Java code. This allows the rule to do more than add a simple annotation to the document representation. In this example rule, each section of the document representation which is matched by the rule is annotated with a `Person` annotation, and the text of the nodes is output for debugging purposes.

5. Saxon Applications

Saxon has been employed as part of a hybrid IE system for the extraction of semantic meaning from descriptions of photos (Carvalho et al., 2008). Fifteen generic rules were developed to identify people, locations and objects mentioned in the descriptions. Extraction using these rules resulted in entities with precision ranging between 70% and 90% and recall between 60% and 76%.

[3]Saxon and full documentation can be downloaded from `http://nlp.shef.ac.uk/wig/tools/saxon/`.

Saxon is also being used within the X-Media project[4] to perform entity and relation extraction from a variety of document formats and structures, making full use of the Runes framework to abstract away the need to be concerned with the orginal document formalism.

6. Discussion

In this paper we have introduced a rule based document annotator called Saxon. Saxon builds upon the open-source Runes framework for document representation. This reliance on Runes allows Saxon to dispense with handling the intricacies of reading from different document formats and leaves rule developers free to concentrate on the task they are trying to perform.

Acknowledgements

This work was funded by the X-Media project (`www.x-media-project.org`) sponsored by the European Commission as part of the Information Society Technologies (IST) programme under EC grant number IST-FP6-026978.

7. References

Arvind Arasu and Hector Garcia-Molina. 2003. Extracting structured data from web pages. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 337–348, New York, NY, USA. ACM.

Javier Artiles, Julio Gonzalo, and Satoshi Sekine. 2007. The SemEval 2007 WePS Evaluation: Establishing a Benchmark for the Web People Search Task. In *Proceedings of 4th International Workshop on Semantic Evaluations (SemEval'07)*, Prague, Czech Republic, June.

Paul Buitelaar, Philipp Cimiano, Stefania Racioppa, and Melanie Siegel. 2006. Ontology-based information extraction with soba. In *International Conference on Language Resources and Evaluation (LREC)*, Genoa, Italy, 5.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1. John Wiley & Sons, Inc., New York, NY, USA.

Rodrigo Carvalho, Sam Chapman, and Fabio Ciravegna. 2008. Extracting Semantic Meaning from Photographic Annotations using a Hybrid Approach. In *Proceedings of the 1st International Workshop on Metadata Mining for Image Understanding (MMIU'08)*, Portugal.

L. Hirschman, A. Yeh, C. Blaschke, and A. Valencia. 2005. Overview of biocreative: critical assessment of information extraction for biology. *BMC Bioinformatics*, 6 Suppl 1.

Jose Iria and Fabio Ciravegna. 2006. A methodology and tool for representing language resources for information extraction. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC'06)*, Genoa, Italy, May.

[4]`http://www.x-media-project.org`

D. Maynard, M. Yankova, A. Kourakis, and A. Kokossis. 2005. Ontology-based information extraction for market monitoring and technology watch. In *ESWC Workshop on End User Apects of the Semantic Web*, Heraklion, Crete.

Einat Minkov, William W. Cohen, and Andrew Y. Ng. 2006. Contextual search and name disambiguation in email using graphs. In *29th Annual International ACM SIGIR Conference*, Seattle, August.

Borislav Popov, Atanas Kiryakov, Damyan Ognyanoff, Dimitar Manov, and Angel Kirilov. 2004a. Kim - a semantic platform for information extraction and retrieval. *Natural Language Engineering*, 10(3-4):375–392.

Borislav Popov, Atanas Kiryakov, Damyan Ognyanoff, Dimitar Manov, and Angel Kirilov. 2004b. Kim a semantic platform for information extraction and retrieval. *Natural Language Engineering*, 10(3-4):375–392.

Lawrence Reeve and Hyoil Han. 2005. Survey of semantic annotation platforms. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1634–1638, New York, NY, USA. ACM Press.

Carlo Strapparava and Rada Mihalcea. 2007. Semeval-2007 task 14: Affective text. In *Proceedings of 4th International Workshop on Semantic Evaluations (SemEval'07)*, Prague, Czech Republic, June.

Xiaojin Zhu, Andrew Goldberg, Jurgen Van Gael, and David Andrzejewski. 2007. Improving diversity in ranking using absorbing random walks. In *Human Language Technologies: The Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*.