Modern amendment drafting - The road to XML

Bert Willems, FontoXML <bert.willems@fontoxml.com>

Abstract

This paper reports on an experiment to build a system which aids in the drafting of amendment documents. The system provides a mechanism to help validate the correctness of amendments. Furthermore, the system is able to semi-automatically sort the amendments in voting order and simulate the effects of amendments on the law. The proposed implementation is based on XML technology, an XML editor and machine learning.

Table of Contents

Context

Disclaimer: The following is a simplified representation of reality intended to provide just enough context to the reader of this paper. It is not intended as a comprehensive introduction to legislation and parliamentary procedures.

New legislation is often introduced by augmenting an existing law rather than introducing an entirely new law. This process is called "amending" the law. An amendment is a formal or official change made to a law, contract, constitution, or other legal document. It is based on the verb to amend, which means: to change. Amendments can add, remove, or update parts of these legal documents.

For example:

3.9 In paragraph A57C(b)(vi), for "his", substitute "their".

This amendment replaces the word "his" with "their" in paragraph A57C(b)(vi).

The process of making a change to the law starts with a deputy (a member of the parliamentary commission and/or the parliament) writing an initial draft of the amendment. This initial draft is sent to the drafting office. The drafting office’s job is to ensure that each legal document (including

amendments) is well written, unambiguous and consistent with the other legal document(s) in the legal system. It is often the case that they make changes to the draft or they need to make changes to other parts of the law in order to keep the legal system consistent. This often requires discussion with the deputy and new versions of the draft go back and forth.

The current process in the parliaments (at least in Italy and the Netherlands) is still very manual: sometimes the amendments are written on paper which needs to be digitalized first. Other times the amendments are written in Microsoft Word. In both cases, the amendments need to be converted before they can be published. The amendments are often published as PDF files or as HTML on a website.

Both the Dutch and the Italian parliament use digital workflows and tools, for example, to register amendments. The content of those systems is still primarily Word-based. Adopting an XML-first workflow would eliminate the need for conversion and would enable a whole new suite of tools. However, introducing changes to parliamentary processes is a time-consuming task, often requiring a change in the statute.

Introduction

This paper reports on an experiment testing whether it is possible to create an XML authoring system which pro-actively aids during the drafting lifecycle of amendment documents, from initial draft to voting. The goal of the system is to provide a user interface (UI) that is quite similar to Microsoft Word but it is XML based. Having an XML-first workflow reduces the conversion burden but that is not something the authors of amendment documents are concerned with. Therefore, in order to get users to adopt the new system, it must provide clear benefits to those users.

The benefits realized during the experiment are:

1. Assisted checking whether an amendment is correct.
2. Ordering amendments for voting.
3. Simulation of the effect of an amendment to the actual law.

The scope of the experiment is to test whether it is possible to realize those benefits in a software system. The first part of the system is a UI where users write the content of an amendment document. The UI is implemented using a WYSIWYG XML editor. The second part of the system provides real-time warnings and the on-demand simulation in case there are no warnings.

Validating amendments

Although the final judgement of whether an amendment is considered valid and correct must be left to humans at all times, the system assists where it can. For example, if a user writes an amendment to change article X in law Y, the system can test whether article X actually exists in law Y and warn the user if it doesn't. The system can also determine whether the basic components (location and action) of an amendment are actually present and give a warning in case they're missing.

Ordering amendments

The goal of ordering amendments is to reduce the amount of voting that needs to happen by considering amendments that have a bigger impact first. The ordering of the amendments is rule-based. The rules depend on the location of the target of the amendment and the amendment action.

Consider the following (abstract) rule set:

1. Amendments are first grouped and sorted by article.
2. Within each sorted group, sort the amendments from those with the biggest impact to the ones with the smallest impact.

For example:

Consider amendments A1 and A2. A1 proposes to change a single word in article 1 and A2 proposes to delete article 1. Since the impact of A2 is much bigger than A1, and in fact "contains" A1, it should be first in the list.

The system should warn the user in case the ordering in the amendment document is different from the ordering prescribed by the rule set.

Simulation

The goal of the simulation is to *simulate the effect* of the change described in an amendment to *the original text*. In order to do so, the system needs to be able to interpret the amendment and apply it to the original text.

Via this simulation, the user gets a real-time preview of the result of an amendment. This enables him/her to see whether the intended effect is actually achieved.

Problem definitions

The system is modeled around four main problems:

Problem 1: Segment amendments in an amendments document

An amendment document may contain multiple amendments. The system must be able to recognize the start & end position of each individual amendment because subsequent problems work on the individual amendment level.

Problem 2: Recognize the location, action and operand information

Each amendment essentially describes a change to the original text. For the next problems, it is required for the system to understand the nature of the change. Essentially, it needs a model of each change.

The model requires:

1. Location information; Where does the change need to be made in the original text.
2. Action information; How does the original text need to be modified.
3. Operand information; Depending on the action information.

This is essentially an information extraction problem.

Problem 3: Order the amendments according to the rule set

The ordering algorithm is defined as a rule set that takes the model recognized by Problem 2 as an input. Essentially this problem can be represented as a topological sort with constraints.

Problem 4: Generate the simulation

The simulation can be modelled as a transformation on the original text taking the result of Problem 2 as an input for the transformation.

Subproblems:

1. Resolve the original text being modified.

2. Find the location in the original text based on the location information.
3. Apply the action and operands.
4. Create a rendition of the result.

Implementation

The architecture of the system is based on the Apache UIMA architecture [Lally and Ferrucci 2004]. It is a component software architecture for the development of analytics for the analysis of unstructured information. This allows the problems introduced in the previous section to be decomposed into sub-problems, each which its own solution forming a system that can be quickly adapted to other problems by swapping analytic components in and out. It essentially forms a pipeline through which information flows. Each stage adds additional information in the form of annotations. The implementation takes XML as an input instead of unstructured text. The annotations can be on both the character level, DOM level as well as on both levels simultaneously.

An example of an analysis pipeline would be to use Regular Expressions to extract information which is then fed into a second stage where a machine learning algorithm could take the matches as input features.

The following sections describe how each previously listed problem is solved.

Segmenting amendments (Problem #1)

Any given amendment document is likely to contain multiple amendments. Sometimes on the same law but is not uncommon to update multiple laws with one set of amendments. Think of an update of multiple laws as a database transaction happening on multiple tables; you want to preserve the referential integrity (or the integrity of the legal system in this case).

The goal of this stage in the pipeline is the recognize where each individual amendment starts and where it ends. The result of the segmentation can be stored in the XML document resulting in a richer XML structure.

Looking at this problem closer, it is composed of two tasks:

1. Distinguish between the preamble, postamble and the body (containing the amendments we care about).
2. Distinguish between individual amendments which are often grouped in articles.

We treat these problems as a sequence classification problem. We trained 2 models implemented using the Conditional Random Fields (CRF) algorithm [Lafferty, Andrew and Fernando 2001]. According to the research performed by Fuchun Peng and Andrew McCallum [Peng and Andrew 2004], CRFs work well for extracting structured information from research papers, achieving good performance. Although amendments are different from research papers, both are generally well-structured and some of the tasks, including segmentation overlap. This algorithm is also used in similar implementations like GROBID [GROBID 2008-2017] and MALLET [McCallum 2002].

Linear Chain Conditional Random Fields (CRF)

We'll treat some of the problems as sequence classification problems. We'll use a well-known algorithm called Conditional Random Fields (CRFs) to solve these problems.

According to Wikipedia:

> CRFs are a class of statistical modeling method often applied in pattern recognition and machine learning and used for structured prediction. CRFs fall into the sequence modeling family. Whereas a discrete classifier predicts a label for a single sample without considering "neighboring" samples, a CRF can take context into account; e.g., the linear chain CRF (which is popular in natural language processing) predicts sequences of labels for sequences of input samples.

> CRFs are a type of discriminative undirected probabilistic graphical model. It is used to encode known relationships between observations and construct consistent interpretations. It is often used for labeling or parsing of sequential data, such as natural language processing or biological sequences and in computer vision. Specifically, CRFs find applications in POS Tagging, shallow parsing, named entity recognition, gene finding and peptide critical functional region finding, among other tasks, being an alternative to the related hidden Markov models (HMMs).

To get a sense of how CRFs work, consider the sentence "I'm at home.". Now consider the sentence "I'm at kwaak.". Based on both sentences one intuitively understands that "kwaak" is some sort of location because we know that "home" is also a location and the words appear in the same context.

CRFs take into account the context in which a word appears and some other features like "is the text made up out of numbers?". More precisely: an input sequence of observed variables X represents a sequence of observations (the words with the associated features which make up a sentence) and Y represents a hidden (or unknown) state variable that needs to be inferred given the observations (the labels). The Y_i are structured to form a chain, with an edge between each $Y_{(i-1)}$ and Y_i. As well as having a simple interpretation of the Y_i as "labels" for each element in the input sequence, this layout admits efficient algorithms for:

1. model training, learning the conditional distributions between the Y_i and feature functions from some corpus of training data.
2. decoding, determining the probability of a given label sequence Y given X.
3. inference, determining the most likely label sequence Y given X.

For a more detailed introduction to CRFs, see An Introduction to Conditional Random Fields for Relational Learning [Sutton and McCallum 2012].

For the implementation of the CRFs, an implementation based on CRFSharp [Fu 2017] is used. CRFSharp is a .NET Framework 4.0 implementation of Conditional Random Fields written in C#. Its main algorithm is similar to CRF++ written by Taku Kudo [Kudo 2017]. It encodes model parameters by L-BFGS. Moreover, it has many significant improvements over CRF++, such as totally parallel encoding and optimized memory usage. The CRFSharp implementation was modified to target the .NET Standard 2.0 to allow cross-platform usage in .NET Core applications.

Model description

The segmentation model is split into two smaller models. The first model segments the amendment text from the surrounding text like introductory and closing words. The second model segments individual articles and the amendments within them.

The most notable extracted features are:

1. Whether the text is inside a heading;
2. Whether the text starts with a list marker, e.g.: 1., a., 1), etc.;
3. Whether the sentence matches a Regular Expression testing for article titles.

The tags used are:

1. preamble
2. heading
3. paragraph
4. amendments
5. article
6. clause (clause of an article, can be nested)
7. postamble

The first model is evaluated on the entire text of the amendments documents. The result of the first segmentation is then fed into the evaluation of the second model which segments the individual articles and amendments within them.

Both models are trained using Conditional Random Fields on a training. The corpus size of the text segmentation model is 35 documents. The corpus size of the amendments segmentation model is 40 documents. Both corpora are drawn from published Dutch amendment documents and tagged by hand. Both models were trained with a maximum iteration count of 1000 using L2 regularization running on 8 threads in parallel. The training of the text segmentation model took 4 minutes while training the article model takes around 6 minutes on a laptop with an Intel i7-4702HQ processor and 16GB of RAM. Training was clearly CPU bound; the 8 logical processors were 100% utilized. Memory usage was around 1.5 GB. Training speed can possibly be improved using the GPU rather than the CPU. However, this was not explored.

The trained models are evaluated against previously unseen examples. Both models are scored on the overall performance of all their labels. Both models are scored using the accuracy and F_1 metrics which are common scoring metrics.

Table 1. Evaluation results

Model name	**Accuracy**	**F_1**
Text segmentation	99,53%	99,72%
Amendment segmentation	95,36%	95,31%

As seen in Table 1, “Evaluation results”, the models are quite accurate. This is mostly due to the predictable and precise nature of the amendment documents.

The result of the evaluation of both models is fed back to the UI where the user sees a suggestion to apply markup an entire article by pressing one button. This UI allows the user to make corrections in case the model did not predict the correct structure.

Recognizing location, action and operand information (Problem #2)

For each amendment, the location, action and operation information must be extracted from the raw amendment text so the system can reason about it. The location information contained details on where the change should be effectuated with respect to the law(s) being modified.

This information includes for example:

- The name of the law;
- The number of the article;
- The number of the clause;
- The position in the text;

The action and operand information are related. The action describes the type of modification. For example, a replacement of a word in the text or the insertion of a new clause. The required operand information depends on the action. For example, a word replacement requires the string to match and the string to replace each match with. A deletion of a clause does not require any further info.

The implementation uses the analysis pipeline architecture again in order to divide the problem into sub-problems. The first stage of the pipeline uses lexicons to detect names of laws and common action words like "substitute".

The second stage of this analysis pipeline is a modified version of Stanford TokensRegex [Chang and Manning 2014]. TokensRegex is a generic framework for defining patterns over text (sequences of tokens) and mapping it to semantic annotations. TokensRegex emphasizes describing the text as a sequence of tokens (words, punctuation marks, etc.), which may have additional annotations, and writing patterns over those tokens, rather than working at the character level, as with standard regular expression packages.

An example of a rule would be:

```
'for' '"' (:<left-op>[]+) '"' ',' [have(app:action)] '"' (:<right-op>[]+) '"'
```

This pattern matches the string: *for "his", substitute "their"*. The pattern consists of literal tokens, ‘for’ and ‘"’ which must be matched exactly. The pattern also contains named capture groups (left-op and right-op) to extract the operands. Finally, the pattern contains a token which must have an app:action annotation. The action annotation is set in the previous stage of the pipeline using a lexicon. The extraction rules were written by hand.

A notable extension made to the TokensRegex syntax is to also allow XPath expressions to be used within the expression. This extension allows rules which take the XML tagging into account. This feature is extensively used to allow users to manually tag something that has not been recognized automatically. Consider the following expression:

```
(:<left-op>'"' []+ '"' | [matches-xpath(‘ancestor-or-self::operand’)]+)
```

This expression would match one or more token which make up either a quoted string -or- tokens which are wrapped in an `<operand/>` element.

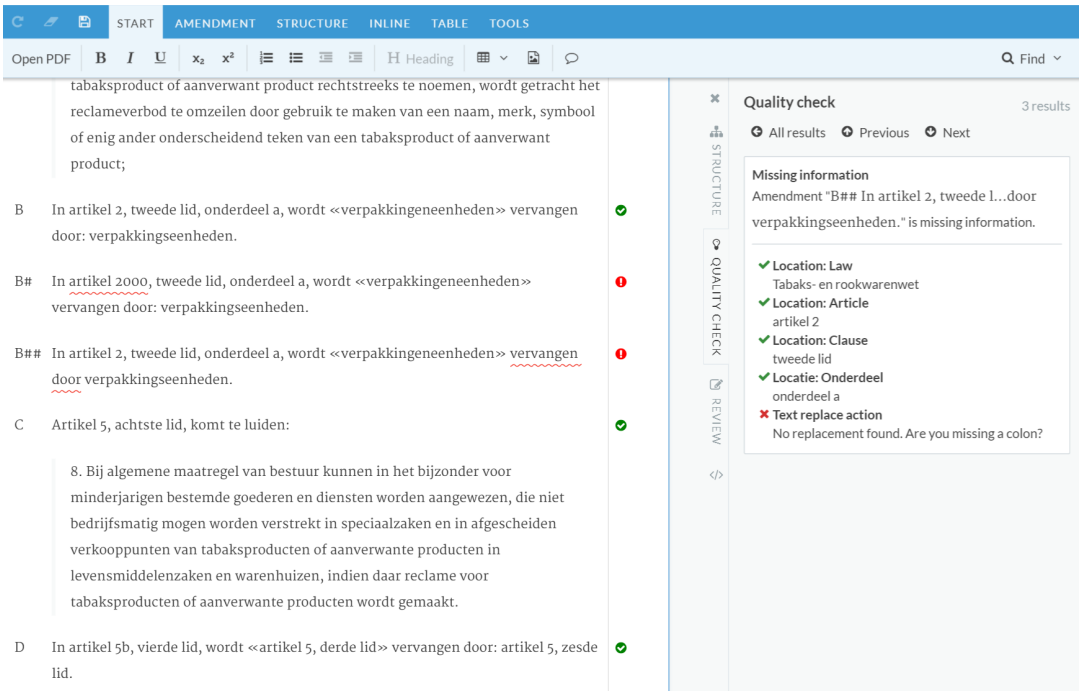
The third stage of this analysis pipeline disambiguates and combines the extracted annotations into an amendment graph representing all the extracted information of each individual amendment. This graph represents both the location information as well as the action information. It is convenient to combine both types of information into a single graph because some of the information overlaps. For example, the word to replace is both location information (which word) as well as an action operand.

The final stage validates the constructed amendment graph. It for example, tests whether the target does actually exist in law being amended. It also tests the amendment for having at least one target and one action. More details of the location validation are provided in the section Simulating the effect (Problem #4).

Validating amendments

In section Recognizing location, action and operand information (Problem #2) it is shown how a graph is constructed and validated. The result of that validation is shown in the UI as well as the graph in the form of a table of extracted information.

Figure 1. Validation warnings



See Figure 1, “Validation warnings”: on the left-hand side you’ll see the amendment document. The amendment in clause B is valid, meaning all the location- and action information was extracted and validated successfully. Clause B# is invalid because article 2000 does not exist in the target law. This is signaled by the red squiggle underline. Clause B## is invalid because it is missing a colon after the words "vervangen door". On the right-hand side you’ll see the details of the warning for clause B##.

Ordering the amendments (Problem #3)

Warning: this section has not been implemented and presents only the conceptual idea of the implementation.

Ordering amendments in voting order is a time-consuming task. The order is defined by a set of rules. This problem can be solved with a topological sort with constraints. The conceptual algorithm is straightforward:

1. Gather all amendments graphs extracted from the amendment document.
2. Create a fictitious root node in the merged graph G_m.
3. For each graph G_a in the gathered amendments graphs:
   a. Add to- or reuse in G_m the location nodes in G_m for each location part in G_a.
   b. Add to- or reuse in G_m the action node.
   c. Add the amendment node to G_m.
4. Assign weights to all the location and action nodes.
5. Sort the amendment nodes based on the weight of the shortest path to the root node.

Once the amendment nodes are sorted it is easy to create the ordered list as the system thinks it should be. The system can then use that ordered list to determine whether it differs from the current ordering in the amendment document and warn the user if they differ.

The tricky part of such an implementation is not the algorithm: it is understanding the ordering rules and defining a weighting scheme based on those rules. These ordering rules are different for each legal system. It may very well be that for some systems it may not be possible to express the rules as a weighting scheme. In all cases, the final order must be left to the user.

Simulating the effect (Problem #4)

Simulating the effect of an amendment to the original text of the law may help users to understand its impact. This is essentially a transformation problem where the transformation is generated from the extracted information in the amendment.

The system performs the following steps accomplish this task:

1. Retrieve the original law.
2. Generate a transformation.
3. Apply the transformation.
4. Show the result to the user.

The first step is to retrieve the original law. Fortunately, the Dutch government publishes the laws as XML documents on the web. For the purpose of this experiment, we downloaded a couple of laws and we implemented a simple web-service to retrieve the XML content of a law based on its name. The name of the law is available in the amendment graph as described in section Recognizing location, action and operand information (Problem #2). This service does not yet take time into account: in the Dutch legal system, laws change over time due to amendments becoming effective on certain dates. Figuring out the effective law is a subject of its own and is out of scope for this paper.

The second step is to generate the transformation. Essentially this step is generating an XSLT stylesheet with a single template. The templates match attribute is generated by creating an XPath expression based on the location information in the amendment graph. The body of the template is generated based on the action and operand information in the same graph. In this step, the amendment graph acts as an Abstract Syntax Tree (AST) which is compiled to XSLT.

The third step is as easy as applying the generated XSLT stylesheet on the retrieved law document.

The final step is to create a rendition of the modified law document. This rendition includes change highlighting to make it easier for a user to see what will be changed.

As seen in Figure 2, “Simulation result”, The UI highlights the changes using familiar change representation: text in red and strikethrough is deleted and green underlined text is added. The yellow dotted border indicates the area affected by the simulated amendment.

Figure 2. Simulation result

The system does not allow the simulation to be stored and it makes it clear to the user that it is a simulation holding no legal value.

Human in the loop

Actual field testing has not been performed at this stage. However, it is already clear that the system will not be accurate in all cases. This is because human language is ambiguous, even in the case of amendments where language is tightly controlled.

In order to not come to a grinding halt, the system is designed to keep the human-in-the-loop (HITL). This effectively means the user and the system work together to accomplish the tasks. The system provides a UI through an XML editor.

If, for example, an operand seems to be missing (e.g. not recognized by the system), the user is provided with a warning. The user can determine whether the required information was indeed missing or whether the information was present but not recognized. In the latter case, the user can manually label the operand by wrapping it in the appropriate XML tag.

Conclusions and further work

The experiment shows that it is possible to build a system that can aid in the drafting lifecycle of amendments using some relatively straight-forward techniques. The system includes a mechanism to validate the correctness of an amendment from a technical perspective. Furthermore, the system aids in the ordering of amendments for voting. Finally, the system provides a method for simulating the effect of an amendment on the actual law.

The main problem is reliable information extraction. Although information extraction algorithms are getting more powerful, there is still noise due to the nature of human language. With XML authoring a hybrid approach can be created where users disambiguate during the authoring process by simply applying markup in case the system gets it wrong. With the addition of XPath to the TokensRegex regular expression language, it is possible to write rules that take manual disambiguation into account.

The described Apache UIMA inspired analysis pipeline architecture works well for this kind of information extraction problems. An observation to make is the described system is essentially a compiler for human language: It takes in the text of an amendment which is essentially lexing and tokenizing. From the tokens, an amendment graph is constructed which is an Abstract Syntax Tree (AST). From the AST an XSLT stylesheet is generated which outputs the modified law. Compiler errors are shown to the user as warnings providing hints that something is not correct just yet.

From a technical perspective, the next step would be to be able to automatically learn patterns. This would make the system scalable across different legal systems without the need of rewriting all the rules from scratch every time. The idea is to augment systems like RAPIER [Califf and Mooney 1997] or WHISK [Soderland 1999] to work with the XML-aware TokensRegex.

So far, we approached this experiment from a technical perspective, although it is based on actual issues encountered in the parliaments today. The next step is to take such a system in production. For a legal system, this would mean writing more rules to deal with edge cases. The biggest hurdle are the legal implications though: some of the features described in this paper may not be used legally in some legal systems due to rules of the parliamentary procedures.

Bibliography

[Califf and Mooney 1997] Califf, Mary Elaine, and Raymond J. Mooney. 1997. "Relational Learning of Pattern-Match Rules for Information Extraction." CoNLL.

[Chang and Manning 2014] Chang, Angel X., and Christopher D. Manning. 2014. TokensRegex: Defining cascaded regular expressions over tokens. Stanford University Technical Report, Department of Computer Science, Stanford University. https://nlp.stanford.edu/pubs/tokensregex-tr-2014.pdf.

[Fu 2017] Fu, Zhongkai . 2017. CRFSharp. https://github.com/zhongkaifu/CRFSharp.

[GROBID 2008-2017] 2008-2017. GROBID. https://github.com/kermitt2/grobid.

[Kudo 2017] Kudo, Taku. 2017. CRF++: Yet Another CRF toolkit. https://taku910.github.io/crfpp/.

[Lafferty, Andrew and Fernando 2001] Lafferty, John D, McCallum Andrew, and Pereira Fernando. 2001. "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data." Proceedings of the International Conference on Machine Learning. 282–289.

[Lally and Ferrucci 2004] Lally, Addam, and David A. Ferrucci. 2004. "UIMA: an architectural approach to unstructured information processing in the corporate research environment." Natural Language Engineering 10: 327-348.

[McCallum 2002] McCallum, Andrew Kachites. 2002. MALLET: A Machine Learning for Language Toolkit. http://mallet.cs.umass.edu.

[Peng and Andrew 2004] Peng, Fuchun, and McCallum Andrew. 2004. "Accurate Information Extraction from Research Papers using Conditional Random Fields." HLT-NAACL.

[Soderland 1999] Soderland, Stephen. 1999. "Learning Information Extraction Rules for Semi-Structured and Free Text." Machine Learning 34: 233-272.

[Sutton and McCallum 2012] Sutton, Charles A., and Andrew McCallum. 2012. "An Introduction to Conditional Random Fields." Foundations and Trends in Machine Learning 4: 267-373.