TextProc – a natural language processing framework and its use as plagiarism detection system

Janez Brezovnik, Milan Ojsteršek

Abstract — A natural language processing framework called TextProc is described in this paper. First the frameworks software architecture is described. The architecture is made of several parts and all of them are described in detail. Natural language processing capabilities are implemented as software plug-ins. Plug-ins can be put together into processes that perform a practical natural processing function. Several practical TextProc processes are briefly described, like part-of-speech tagging, named entity tagging and others. One of those is capable to perform plagiarism detection on texts in Slovenian language, which is explained in detail. This process is actually used in digital library of University of Maribor. The integration of digital library with TextProc is also briefly described. At the end of this paper some ideas for future development are given.

Keywords—natural language processing, text processing, text mining, plagiarism detection, software framework, Slovenian language

I. Introduction

TextProc (abbreviation of "text processor") is a natural language processing framework. The "natural language processing" means that it is intended for processing of texts, written in human language. The "framework" means that the software is intended to be used as a building block for other software applications that require natural language processing capabilities. TextProc was implemented at Laboratory for heterogeneous computer systems, Faculty of Electrical Engineering and Computer science of University of Maribor. Natural language processing capabilities are implemented as software plug-ins. Because of this, the framework itself is language independent, since all language dependencies are limited to the plug-ins.

There are two main ideas for its implementation. First, natural language processing is implemented in the form of software plug-in once and then reused many times. An expert in a specific natural language processing field implements a plug-in. Later, other users can use it without knowledge of its implementation. Plug-ins are also developed in such a way, that it is possible to put them together into processes (we call them TextProc processes), which perform a more complex natural language processing operation. Second, the framework assures an easy way to build and execute before mentioned processes and also makes them available through several software interfaces for integration. This way, TextProc processes can be used in other applications, which make TextProc even more useful. Some ideas are based on a similar natural processing framework, called GATE (General Architecture for Text Engineering) [1][2].

TextProc is implemented in C#, based on Microsoft .NET Framework 1.1. The TextProc framework itself and some plug-ins also use Microsoft SQL Server 2000 database or better, although plug-ins can use any kind of database or other data storage method. Output for TextProc is mostly in XML format, so XSLT (XML transformation) together with web technologies like XHTML, CSS and JavaScript are used.

The software architecture of TextProc framework is described in the second chapter, together with detailed descriptions of all modules in the architecture. Third chapter introduces to TextProc plug-in types and explains the inner working of a plug-in. Forth chapter presents the structure of a TextProc document. It is used for intermediate and final results. It is sent among plug-ins and defines the only way plug-ins communicate with each others. Plug-ins can be put together into processes that perform a practical natural processing function. Details about TextProc processes are explained in fifth chapter, where some practical processes are briefly mentioned. One such process is plagiarism detection, which is explained in sixth chapter. Concluding remarks are done in the last chapter.

II. Architecture

TextProc framework is made of several modules, as is shown on Fig. 1. If we go from the bottom up, we first see the TextProc plug-ins. TextProc knows five types of plug-ins, where only one type is actually meant to implement natural language processing algorithms. Other plug-in types perform reading and writing operations on various data sources, like databases, web pages and file systems. TextProc plug-in types are described in detail in the next chapter.

Plug-ins must be implemented according to the software interface that is defined in second module, called ITextProc, where the letter "I" stands for "interface". ITextProc also defines the TextProc document object that contains the plain text and all other data about the document, which is processed,

including all results that plug-ins produce. This document is then passed among the plug-ins. TextProc document is explained in details later. ITextProc module also includes all the functionality that has proven to be generally useful when implementing plug-ins, like precise timers, HTML code manipulation, simplified functions for hashing algorithms, XML transformations, reading and writing files and more.

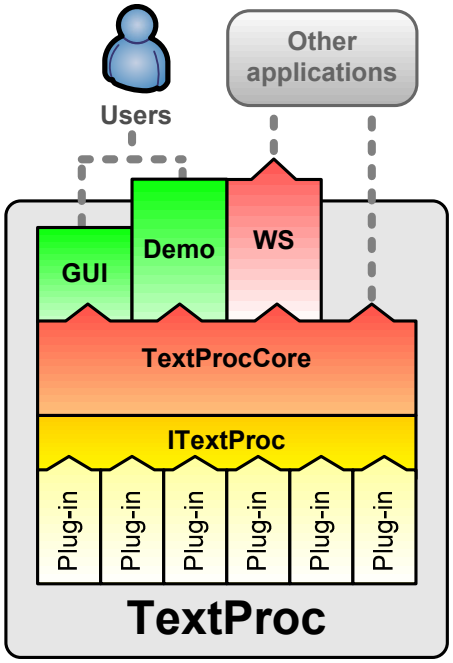

Fig. 1: TextProc architecture

Both ITextProc and plug-ins are implemented as .NET assemblies, also known as DLL files (Dynamic-Link Library). ITextProc must be used by plug-in developers for proper implementation. When a plug-in is developed and compiled, the resulting DLL is copied into a special folder of the TextProc framework. If the plug-in is properly implemented, the framework loads it and makes it available for use in TextProc processes.

TextProcCore module implements everything in TextProc framework that is not implemented in already mentioned modules or it isn't a part of GUI (Graphical User Interface) directly. Its main purpose is to manage and run TextProc processes that are made of plug-ins. This module (together with ITextProc and plug-ins) can also be used directly for integration with 3th party applications, if they are capable to load and run .NET assemblies. With just a few lines of code a developer can load and run a selected TextProc process and receive the result. Results can be in the form of the TextProc document object or as XML. Since XML can be transformed using XSLT, the end result that the developer receives can be almost in any form, making development of 3th party applications as easy as possible. This presents the first possible method of integration.

Previously mentioned modules are enough to run TextProc processes, but to use TextProc directly by the user it requires additional user interfaces. TextProc provides two user interfaces: a desktop application and a web application.

TextProcGUI is a desktop application and is the main user interface for TextProc. Its primary purpose is to enable creation, management and execution of TextProc processes. It is intended for execution of long running processes on a large collection of documents. It also enables management of processes, available via TextProcDemo and TextProcWS (described later). The main window of the TextProcGUI is shown on Fig. 2. In the "Log" section (bottom) the event log is shown, where exceptions and other information is displayed, like how long it took for process to finish. Some plug-ins also write their statistics at the end of processing in the log window.

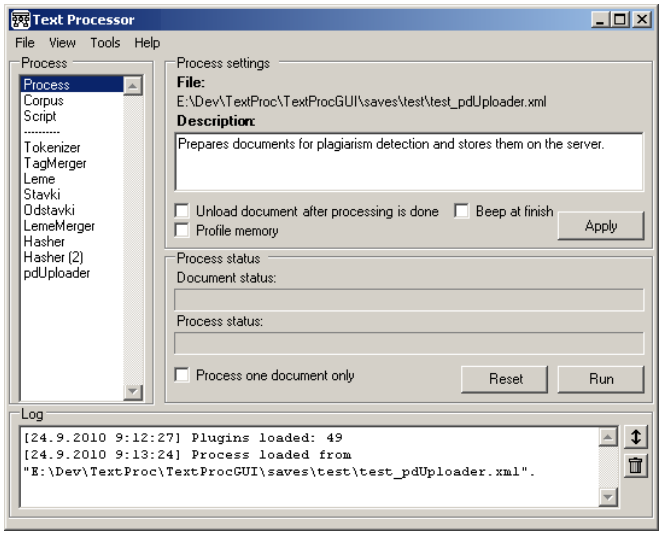

Fig. 2 Main window of TextProcGUI

In "Process" section a list is shown, where the first three items are always shown: process, corpus and script. If the first item is selected, then some basic settings of TextProc process are shown (on the right of the list). From here TextProc process can be started (with "Run" button) and the progress can be observed. There are two progress bars: for progress of the current document and for the entire corpus. If the "Corpus" list item is selected, we get a list of documents that are about to be processed. Documents can be loaded from the file system or from various databases; currently Microsoft SQL Server 20xx and MySql databases are supported. If the "Script" list item is selected, we get a multiline text input field, where a simple domain specific scripting language can be used. The script is explained in detail in later section.

All other list items in the "Process" list (below the line) are plug-ins. Selecting one of them shows all settings of the selected plug-in. Plug-ins can be added to the list and thus to the TextProc process by right clicking on the list and selecting a plug-in. A single plug-in can be added to the process multiple times and each instance can have different settings. Settings of a plug-in are actually all public properties of a C# class that implements the plug-in interface. For now only primitive data types are allowed.

TextProc is also available over the web in two parts. First part is a web application called TextProcDemo that is primarily intended for presenting TextProc capabilities. It is also used for validation of correct implementation of plug-ins and processes. Some TextProc processes are published and publicly available to everyone for testing purposes. Web page for testing is shown on Fig. 3. TextProcDemo is currently available only in Slovenian language; Fig. 3 was translated for this paper.

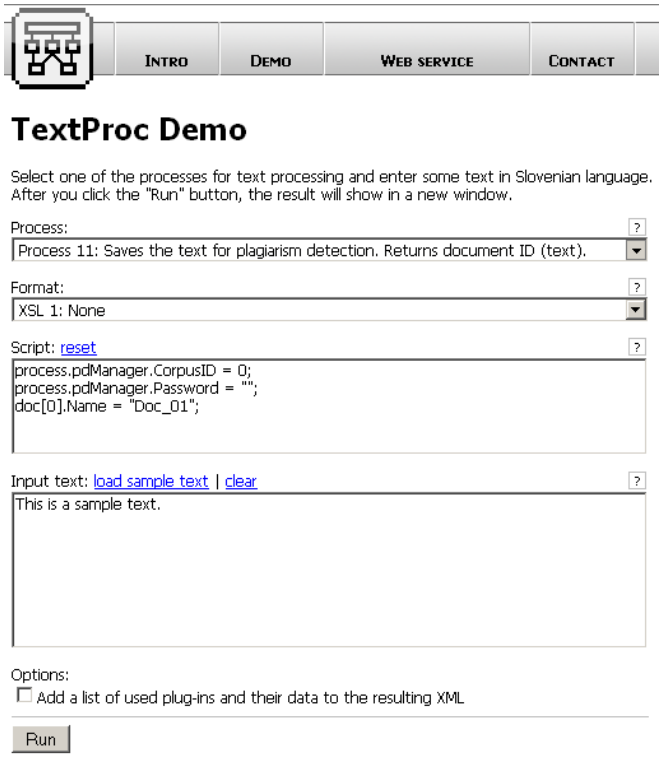

Fig. 3: TextProcDemo

First, the user selects one of the available TextProc processes. At the end of short description (in the dropdown list) the output format is mentioned and mostly the output is in XML format. XML can be transformed using XSLT, so the next dropdown list includes all available transformations for the selected process. Since TextProcDemo is intended for testing purposes, most transformations produce a human readable output, thus in XHTML format. The third input expects commands in the form of a script. Most processes don't require a script, but some do. If the script is required for process execution, then those lines of script (with default values) are displayed when the process is selected, as is also shown on Fig. 3. This way the user knows that the script is required. The forth input is for the text, that is about to be processed. There is also an option to add data about used plug-ins into the resulting XML. Full name and version of used plug-ins is added, including execution times for each plug-in, which can be used for plug-in performance comparison. Once all the data is entered, we can press the "Run" button and the results are shown in a new browser window. This way the process settings and entered text remain entered and enable the user to make small changes to settings or text for another run. Since each result opens in a new window, it is also possible to compare results of different process executions.

The second part of TextProc that is accessible on the web is TextProcWS. This is a .NET web service that can be used to run exactly the same TextProc processes as TextProcDemo. Input parameters are also the same, where process and XSLT file are determined by the identification numbers. Users can get those numbers from dropdown lists on TextProcDemo web page. In case of Fig. 3, the process identifier is 11 and XSL file identifier is 1 (no XSL selected, result will be in XML format).

III. TextProc plug-ins

TextProc plug-ins are modules, that carry the natural language processing and text mining functionality. They enable the TextProc framework to grow. However TextProc supports multiple types of plug-ins and only one type is actually involved into natural language processing while others are in a support role. Currently, TextProc supports five types of plug-ins:

- InputPopulator,
- InputConnector,
- Converter,
- Processor,
- OutputConnector.

The main purpose of the InputPopulator is to populate the corpus of a TextProc process with documents. TextProc corpus can include documents from all kind of sources, but each source has a different way of access. This plug-in type takes care of content delivery of selected documents, depending on the source. For instance, getting document from a file system requires from a user to define the folder and file type (file extension), while getting document from a database requires a connection string and a SQL query. This plug-in type is used only in TextProcGUI, when a user is preparing the process corpus. Currently TextProc has 4 plug-ins of this type: for getting document from a file system, from a Microsoft SQL Server 20xx database, from MySql database and for filling the corpus with empty documents. When the corpus is filled with documents, it is filled with instances of the next plug-in type, called InputConnector.

For each InputPopulator plug-in there is an InputConnector plug-in; they always appear in pairs. When the corpus is populated with documents, it is actually filled with instances of InputConnector plug-ins. Each instance carries all the data necessary to read the content from the document source. E.g. for reading a file from file system, there is an InputConnector instance that carries a single file name; for reading the content from a database, there is an InputConnector instance that carries a SQL query, which returns the text and so on. Real implementations of these plug-ins in fact carry more data. As said, instances of this plug-in type are stored in corpus and if

the process is saved, then the plug-in settings are stored in the file; not the actual document content, but the data required for content access. Process is saved as XML file and can be viewed and edited in any text editor. A sample process XML with only one document in the corpus can be seen on Fig. 6 (on next pages).

Since InputPopulator and InputConnector plug-ins always appear in pairs, there are currently 4 plug-ins of type InputConnector: for file system documents, for Microsoft SQL Server database records, for MySql database records and for empty documents. The last type is used in TextProc processes, available at TextProcDemo web site. These processes include an empty document that is filled with text, entered into the input box.

The third plug-in type - called Converter - is intended for converting various document formats into plaintext. TextProc is only capable of processing plaintext, so documents formats like PDF or DOC must first be converted to plaintext. At the time all input was externally converted to plaintext, so there a currently no plug-ins implemented of this type, since there was no need for this. The idea was that plug-ins of this type would register itself into TextProc as a converter for a specific document MIME type, but such functionality was never implemented.

The most important plug-in type in TextProc is Processor. This plug-in type is actually involved with natural language processing and also text mining. Currently there are 39 plug-ins implemented of this type. Processor plug-ins can be added to the TextProc process or removed using TextProcGUI. They are stored (with settings) as parts of the process XML file (as shown in Fig. 6 on later pages). Plug-ins of this type are also accessible via a script, that enables us to change plug-in settings while the process is running. Using the script it is also possible to skip a plug-in execution, thus disabling the plug-in (or enable an already disabled one). More about the script will be explained later.

The last type of plug-in is OutputConnector. It behaves exactly the same as plug-in type Processor and is intended for writing results. Plug-ins of this type are mostly used at the end of a TextProc process, where results are written. For now there are 5 plug-ins implemented of this type. In most cases, results are written in XML format, since XML is excellent for software interoperability and enables easy output formatting later on (using XSLT). Other plug-ins of this type return specific result formats that can't be achieved using XSLT or return statistics, not included in the XML file.

Implementation of a TextProc plug-in is quite easy. Fig. 4 shows the minimum source code (in C# programming language) that successfully compiles as a TextProc plug-in, although the given code doesn't do anything. As seen, a TextProc.ITextProc namespace must be used and the class must be implemented according to IProcessor interface. The class name must be the same as the plug-in name, that is the same as the resulting DLL file. DLL file can include other classes, but only the class with the same name as the DLL file is considered as the plug-in implementation class. This class must have two class properties. The first is *ProcEnv* that is set by the TextProc process and includes a reference to processing environment. This enables access to the entire corpus, event log, progress indicators (on GUI) and other functionality. The second property returns the type of the plug-in (this property it is read-only). Any other properties defined in the class are considered to be settings of the plug-in. Those are displayed in TextProcGUI, can be changed via GUI or TextProc script and are stored as part of the TextProc process XML (shown on Fig. 6). For now, only primitive data types can be used for class properties.

```
using System;
using TextProc.ITextProc;

namespace TextProc.Plugin {
  /// <summary>Sample TextProc plug-in.</summary>
  public class SamplePlugin: TextProc.ITextProc.IProcessor {
    private TextProc.ITextProc.IProcEnv _ProcEnv = null;

    #region Properties

    /// <summary>Processor environment.</summary>
    public TextProc.ITextProc.IProcEnv ProcEnv {
      get{return _ProcEnv;}
      set{_ProcEnv = value;}
    }

    /// <summary>Plug-in type. Read-only.</summary>
    public TextProc.ITextProc.PluginType PluginType {
      get{return TextProc.ITextProc.PluginType.Processor;}
    }

    #endregion

    /// <summary>Default constructor.</summary>
    public SamplePlugin() {
    }

    /// <summary>Main plug-in function.</summary>
    /// <param name="doc">Document for processing.</param>
    public void Run(TextProc.ITextProc.Document doc) {
    }


    /// <summary>Called at the end of the process.</summary>
    public void Finish() {
    }
  }
}
```

Fig. 4: Minimum source code of a TextProc plug-in

The rest of the code includes three methods. First is the default class constructor. Second is the method *Run*, which receives the current document that is about to be processed. This method does the actual processing and is called for each document of the corpus separately. When all documents are processed, then the method *Finish* is called. This method is intended to do any final work of the given plug-in, like storing cumulative results or showing statistics of the entire corpus.

If we have a large set of plug-ins and each having its own settings and meaning, it is important to document them well. For this purpose TextProc supports several ways to document plug-ins. The preferred format for plug-in documentation is HTML, but plaintext files are also supported. The majority of

documentation is stored inside CHM file (Microsoft Compiled HTML Help; it is simply a collection of HTML files stored inside a single file). Documentation of plug-ins can be stored as:

- a HTML file, stored inside the CHM file;
- a HTML file, stored in the same folder as the plug-in and with the same name, except the file extension (.html) or
- a TXT file, stored in the same folder as the plug-in and with the same name, except the file extension (.txt).

At the start of TextProcGUI desktop application, plug-ins are loaded. For each loaded plug-in, the presence of the documentation file is also checked. If documentation for a specific plug-in is available, it is also shown on user request. Practice has shown that the documentation for a TextProc plug-in must include:

- description of what the plug-in does;
- a list of all settings with description of their meaning and allowed values;
- a list of other plug-ins this plug-in depends on;
- description of any external files that plug-in receives via settings.

Documentation for all plug-ins and the rest of the TextProc framework is available in TextProcGUI via help options in the menu and via help button at the plug-in settings screen (for that specific plug-in). Documentation is also available on the first page of TextProcDemo web site.

IV. TextProc Document

The idea of plug-ins in TextProc is that each plug-in performs some work from areas of natural language processing or text mining, using the results of other plug-ins. This requires some kind of communication between the plug-ins. This is done through a special object we call TextProc document. It is a document structure, that caries all the results of the plug-ins. When a TextProc process is executed, the used plug-ins are executed in the given order, as is determined by the process creator (user). At the start, TextProc document is created and filled with plaintext. Plaintext can be read from all kinds of sources and a plug-in of type InputConnector is responsible for delivery of plaintext from a specific source. Plug-in of this type is a part of TextProc document and not of the TextProc process.

After the document object is filled with text, the first plug-in in the process is executed (the method Run in called). It does its job and saves its results in the TextProc document, which is then passed to the next plug-in and so on to the end of the process. Each subsequent plug-in can use the results of the previous one. The last plug-in in the TextProc process is usually of plug-in type OutputConnector, which is responsible for storing the results. In most cases results are saved as TextProc document in XML format. This can be written to a file or to memory for further processing, as is the case when using TextProcDemo or TextProcWS.

TextProc document contains the following data:

- Reference to the InputConnector plug-in that delivered the plaintext content. As already mentioned, there are multiple types of TextProc plug-in and one type is responsible for plaintext delivery.
- Generic key-value data structure for storing various metadata. Plug-ins can add on change this data. For example, processing times of plug-ins are written to this data structure (if requested). It is also used for plug-in communication, when the data in question is not a part of the result, like passing parameters and settings.
- Actual content as plaintext.
- Sets of tags. This stores the actual language processing results.

Sets of tags are used for data exchange between plug-ins. TextProc document can carry any number of sets that are identified by name. Each set can carry any number of tags. A tag is an object that points into the text and carries a set of values (strings). Tag object is defined in ITextProc module and contains the following data:

- Tag identifier (a number), unique per document.
- Position of first character in the plaintext that is tagged by this tag.
- Position of the last character in the plaintext.
- Length of tagged text, calculated from positions of the first and last character.
- Line number of the first tagged character.
- A set of child and parent tags. A tag can point to other tags or be pointed to by other tags. This enables building hierarchies like: a word is a part of a sentence and a sentence is a part of a paragraph.
- A key – value collection of strings.

Fig. 5 shows an example of TextProc document after the TextProc processing is completed. First we have a sample plaintext. One of the first TextProc plug-in in the TextProc process is usually a tokenization plug-in that breaks text into words. This plug-in creates a set of tags, called Tokens. Each set can contain any number of tags. Each tag points into the text, that is on the first and last character of the string that is tagged by a given tag. Each tag can carry any number of properties and in our example it carries the property called TC (token class) and a value. The current TextProc document is then send to the next plug-in, responsible for detection of clauses. It creates a new set, with tags that carry the type of the clause. However this newly created tag doesn't point directly at the text, but on previously created tags with words. Again the document is sent to the next plug-in and this time it tags sentences in form of a new set (called Sentences) and with a new tag that points to previously created clause tag. This way a hierarchy of tags is created, which can be used by any subsequent plug-in. As described, the TextProc document is send to each plug-in in the TextProc process. Usually the last plug-in in the process is of the type OutputConnector that is responsible for writing the results. In most cases the whole

TextProc document is serialized as XML file.

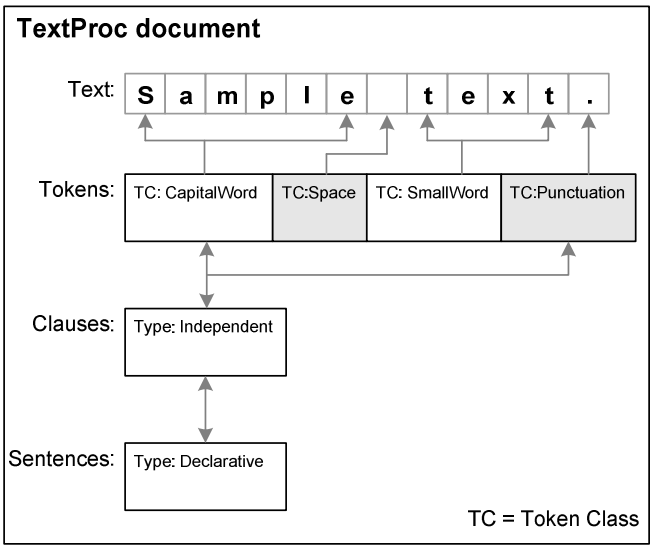

Fig. 5: Sample TextProc document after processing

Since tags are in named sets, and values are in a key-value collection, those names and keys must be known in advance. Because of this, plug-in implementation somewhat depends on implementation of other plug-ins. This means that the order of plug-ins in a TextProc process is predetermined. For now, correct order of plug-ins is not determined automatically and must be taken care of by the user, who creates the process. To somewhat mitigate this problem it is a good idea to give these names as part of plug-in settings. This makes plug-ins more reusable.

V. TextProc process

As already mentioned, TextProc plug-ins aren't executed alone as a single unit, but as a part of context we call the TextProc process. This process can be created using TextProcGUI desktop application and consist of several components:

- a set of plug-ins used in the process,
- a set of documents to be processed (a corpus). Documents can come from various sources and a TextProc process can process them regardless of the source.
- TextProc process settings.
- A script.

In the TextProc process, plug-ins must be in a specific order and the user that is creating the process must know what the correct order is. Each plug-in also has its settings that are stored as part of the process.

TextProc process itself also has its settings, also shown in the TextProcGUI main window (Fig. 2 on previous pages). Those settings are:

- a short description of what the process does. It is a good idea to mention the output format.
- An option to enable or disable removal of already processed documents from memory. While TextProc process is running, TextProc documents are stored in computer memory. If the plug-ins need only the current document for processing, then there is no need to hold the previously processed document in memory.
- An option to enable or disable memory profiling. If enabled, then memory usage measurements for each plug-in is added to the metadata of the TextProc document. There is an OutputConnector type plug-in that can write the memory usage report as HTML file. Measurement of memory usage has a negative impact on execution times of TextProc process and is therefore enabled only if those measurements are needed.
- An option to make a beeping sound when the process is completed. Useful when processing large corpuses on a separate machine. This way the user is notified by a short beep using computer system speaker. Useful only if the process runs for a few hours, but not when running for days. In this case a notification via email would be more appropriate (it is not implemented).

TextProc script is a domain specific language that enables us to make changes to settings of used plug-ins and to manipulate with the process while the process is running. We can:

- change plug-in settings;
- enable or disable a plug-in. If a plug-in is disabled, it is skipped from any subsequent process executions;
- reset, start or end a process;
- store content of the event log to a file or clear the log;
- store and load a TextProc process. Loaded process is then executed automatically.

With the script it is possible to automate experiments, where a single process restarts itself multiple times with prerecorded changes to plug-in settings. Such processes can then run for days without human intervention.

When a created process is stored, it is written in XML format. Fig. 6 shows a sample TextProc process. It has an empty script, one document in a corpus (content is read from a plaintext file) and two plug-ins, Tokenizer and XmlWriter. Each plug-in has its settings. Since TextProc processes are stored in XML and since XML is just a kind of plaintext, they can also be edited with any regular text editor.

There are already many practical TextProc processes available for use. The TextProcDemo web application alone has 14 published processes, so let's mention some of more practical ones.

One process is capable to tag known named entities in Slovenian language. Known entities are stored in a database and it includes Slovenian personal names (first and family names), towns, cities, rivers, hills, mountains, colors, names of days and months.

```
<?xml version="1.0" encoding="utf-8"?>
<Process Beep="False" ProfileMemory="False">
  <Desc>Sample TextProc process.</Desc>
  <Script><![CDATA[]]></Script>
  <Corpus>
    <Doc>
      <Connector Plugin="FileConnector">
        <FileName>D:\input\sample.txt</FileName>
      </Connector>
    </Doc>
  </Corpus>
  <Plugins>
    <Tokenizer Title="Tokenizer">
      <WordClassesFile>%tpRoot%\wcf.txt</WordClassesFile>
      <Chars />
      <Delimiters />
    </Tokenizer>
    <XmlWriter Title="XmlWriter">
      <Path>D:\output\</Path>
      <AddTagValueOfTagSet>Tokens</AddTagValueOfTagSet>
      <XslFile />
    </XmlWriter>
  </Plugins>
</Process>
```

Fig. 6: Sample TextProc process in XML

There is also a TextProc process for tagging numbers, written as words or as numerals. For instance, both "3" and "three" are tagged. In both cases the tag also carries the numeric value, for instance "twenty-three" has a value of 23. It is capable to detect numbers written in both Roman and Arabic numeral systems, also dates and other formats, written with numerals.

Another useful TextProc process is a part-of-speech (or PoS) tagger for Slovenian language. We didn't implement our own tagger; we integrated and existing one, a language independent, freely available tagger, called TreeTagger [9]. This tagger must first be trained on a corpus and we used two Slovenian corpuses, ELAN [7] and later also FidaPlus [8]. We actually used TextProc to convert these two corpuses into training corpus for the TreeTagger training component. The TextProc process using TreeTagger is now also available via TextProcDemo for testing and via TextProcWS for use.

TreeTagger is also able to perform lemmatization. Before TreeTagger was available, we had a different approach to lemmatization, now also available as a TextProc process via TextProcDemo. Since we now have several ways to do lemmatization, we created a special TextProc process that performs lemmatization with all available methods we have and display the results in a single table for comparison. This process is also available via TextProcDemo.

There is also a process published at TextProcDemo that is capable to perform a basic kind of wikification. Wikification is tagging the input text with links to Wikipedia, in our case the Slovenian version. It is possible to download the entire Wikipedia database and we did this for Slovenian an English version and then restored both databases on our servers. Current implementation is basic in the sense that it tags all words available in Wikipedia, which is not very useful (too many links, also on basic words and numbers). In the future it will tag only relevant words, depending on the subject of the input text.

TextProc is also capable to perform plagiarism detection and it takes two TextProc processes to do this. A detailed description of how it works is given in the next chapter.

VI. Plagiarism Detection

One of the most useful practical applications of TextProc for now is plagiarism detection. It is also the only TextProc process currently used by external software (described later). Plagiarism occurs, when someone copies some content from other authors and then claims it as his own original work. Plagiarism is stealing and is therefore illegal. Plagiarism detection is a method of finding content that has been copied from others. A more detailed description of plagiarism, why it happens and the problem it causes is available in [4] and [5].

Plagiarism detection in TextProc is implemented using two TextProc processes. The first process transforms all documents into a form, more suitable for plagiarism detection. This transformation is performed using the following TextProc plug-ins (each list item is a plug-in):

1. Text is tokenized, that is broken into words. This is a generic plug-in and breaks the content up regardless of its meaning. Because of this, certain content is broken, that from a human perspective should not be, but this problem is then solved by the second plug-in.
2. Some tokens are merged back together by a given set of rules. For instance, a decimal number "3.14" is separated by the first plug-in. The second plug-in determines that the dot is not a sentence separator, so it is merged back into one word.
3. All words are converted into lemma form (canonical or dictionary form of the word). This is the most language specific plug-in in the whole process. Lemmatization is used because Slovenian language is heavily inflected; a word can have a very different form, depending on gender, case and number of the word.
4. Sentences and clauses are determined.
5. Paragraphs are determined.
6. Words in lemma form are merged into new sentences without redundant spaces, tabs or line feeds that may be present in the original text; only a single space character is used as word delimiter. Also, words are sorted alphabetically on the level of a clause. This way, word order within clauses becomes irrelevant.
7. Newly constructed sentences are hashed using a hash algorithm. Currently MD5 (Message-Digest algorithm 5) is used; several variants of SHA algorithm (Secure Hash Algorithm) are already supported.
8. Previous plug-in in called again; this time it hashes whole paragraphs.
9. Plaintext of documents and its hash values for sentences and paragraphs are stored in a database.

This process is executed for each document separately. The second process contains only one TextProc plug-in and all it does is searches for hash values from one document that are also present in other documents of the same corpus. The end

result is similarity report in XML format. Similarity is calculated as quotient between the length of similar content and length of entire document, expressed as percentage (for both documents).

Plagiarism detection is currently been used by the Digital library of University of Maribor (DKUM). TextProc has been integrated with DKUM using web service, available as part of TextProcWS. Each night (in times of least web traffic) DKUM sends unprocessed documents to TextProc and requests similarity reports in XML format. Those are then processed and saved in DKUM's database for later use. Reports are saved in such a way that it enables progressive plagiarism detection. This means that a similarity report for a new document updates all reports of the older documents. Similarity reports are then shown on the administrative pages of DKUM, described in detail in [3].

Till now, plagiarism detection is done only between documents in DKUM corpus. In future, other sources will be added. This will probably bring the need to compare document between corpuses, which is now missing. Current implementation of plagiarism detection also lacks the ability to determine citations that are allowed and are thus completely legal. We are still working on these improvements, since they are essential feature for good plagiarism detection.

There are also other ideas for improvements. For instance we would like to detect numbers in text and replace them with a text or tag like "[number]". For plagiarism detection, actual numeric values are irrelevant. If someone does plagiarism deliberately, then they probably make small changes in numbers, like adding a decimal, removing it, maybe changing it or writing it differently. Replacing the number with a tag makes such changes irrelevant and detectable. We already developed a plug-in that is able to tag all kind of numbers. All that we need now is to replace those numbers with a tag. This replacement would be done somewhere before plug-in number 6 (new sentence creation) in the first TextProc process for plagiarism detection.

VII. Conclusion

The existing capabilities of the TextProc natural language processing framework were presented, including practical applications of it. One of the first changes in TextProc will be to port it to the new Microsoft .NET Framework version. It is currently based on .NET 1.1, but the latest version is already 4.0. There are many new features in the latest .NET version and also in the C# programming language that we use. By porting to the latest version we expect both simplifications in software development and performance gains. The latest .NET version also brings significantly better support for parallel computing, that might be useful for plug-in development or in the TextProc framework itself.

TextProc has already seen some practical usage, especially in the role of plagiarism detection system that is integrated into an actual digital library. There are ideas to implement a separate web application that would offer plagiarism detection service to anyone, again based on TextProc. Actually this is already possible by using TextProcDemo, but the current user interface in generic for all TextProc processes and is thus not appropriate for uploading large number of documents. A specific web application will be implemented, including registration, login, document management and report review functionality, targeted at average users (non natural language processing experts). There are also other existing applications that we were already thinking to enhance them with TextProc. One of them is a 2.0 version of proprietary question answering system, currently in development (the first version is described in [6]).

TextProcDemo has also proven to be very useful for testing and presentation purposes, since it is easily accessible (over the internet; no software installation is required). It is also easy to use; users only need to select a process and enter some text. Although results are mostly in XML format, those can be transformed using XSLT into almost anything, including into good looking, human readable format and some of those are already available for use. With further plug-in development it is easy to extend the capabilities of TextProc and its use in the future.

References

[1] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan. "GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications", Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02). Philadelphia, July 2002.

[2] "GATE: General Architecture for Text Engineering", http://gate.ac.uk/, visited on October 2010.

[3] J. Brezovnik, M. Ojsteršek: "Advanced Features of Digital library of University of Maribor", International Journal of Education and Information Technologies, NAUN, Issue 1, Volume 5, 2011, pp.34-41.

[4] S. Carmen Cismas: "Anti-Plagiarism Strategies for Environment Engineering Students", Recent Advances in Energy & Environment, Proceedings of the 5th IASME / WSEAS International Conference on Energy & Environment (EE'10), 2010.

[5] Z. Mahmood: "Students' Understanding of Plagiarism and Collusion and Recommendations for Academics" WSEAS Transactions on Information science and applications, Issue 8, Volume 6, August 2009.

[6] I. Čeh, M. Ojsteršek, "Developing a Question Answering System for the Slovene Language", WSEAS Transaction on Information science and applications, Issue 9, Vol. 6, 2009.

[7] "Slovene-English Parallel Corpus IJS – ELAN", http://nl.ijs.si/elan/, visited on December 2010.

[8] "FidaPlus: Corpus of Slovenian language", http://www.fidaplus.net/Info/Info_index_eng.html, visited on December 2010.

[9] H. Schmid: "TreeTagger - a language independent part-of-speech tagger", http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/DecisionTreeTagger.html, visited on December 2010.