Uplug – a modular corpus tool for parallel corpora

Jörg Tiedemann

Department of Linguistics, Uppsala University

Abstract

This article describes the Uplug-system*, a modular software platform intended for the integration of text processing tools. It includes three components: An extensible I/O library which provides a transparent interface for working with textual data, a tool for combining single-task text and corpus processing modules into sequentially executable systems, and a graphical user interface for running Uplug applications, modifying parameter settings, and investigating resulting data. The system supports a variety of storage formats including those of standard database management tools such as SDBM and GDBM as well as simple XML formats and other text oriented data formats. Furthermore, connections to relational databases are supported via a transparent database toolbox. Uplug applications can be adjusted easily by modifying standardised configuration files. A prototype of the Uplug-system is currently used in a Linux version at Uppsala University with modules for processing bilingual parallel text, such as modules for several kinds of word alignment and data generation from parallel texts as well as tools for the examination and evaluation of the results that are produced.*

1. Motivation and Background

The Uplug-system was developed at Uppsala University within the on-going PLUG project. The project's aim is to develop, evaluate, and apply approaches to generation of translation data from bilingual text (Ahrenberg et al. 1998).

The project is based on former studies on the extraction of translation equivalents, which were carried out at the department of linguistics in Uppsala (Tiedemann 1997, 1998). Henceforth, these studies will be referred to as *LexEx* study. A set of tools and approaches for the work on textual data was implemented. The LexEx study was based on investigations on Swedish, English and German parts of the Scania95 corpus (Scania corpus homepage). The Scania95 corpus comprises a collection of technical documentations in 8 European languages, which were provided by the Scania CV AB in Södertälje/Sweden in 1995. The documents were converted to TEI-conformant SGML (Tjong 1996a) and automatically aligned on the sentence level (Tjong 1996b). Several command-line oriented tools were developed in order to process and to query the Scania95 corpus. For this purpose, a number of intermediate storage formats were created for efficiency reasons. In the LexEx study an Uppsala-specific data format for sentence-aligned bilingual texts were applied for further investigations (Uppsala align format). Several approaches for the extraction of translation equivalents were developed which were applied independently to the text collection. Each approach applied several storage formats for the storage of intermediate results. Finally, bilingual

lexicon files were compiled as the result of the extraction process. These data could be merged and presented in different forms by data conversion tools. Conversion tools were developed in order to merge these data and to present them in different formats.

With the start of the PLUG project, the co-operation between Linköping University, the University of Gothenburg, and Uppsala University was initiated. In the first stage, a common project corpus was established and aligned sentence-wise. Each partner contributed parts of the corpus. The contributions were encoded in different formats depending on the internal standard that were used at each partner's site. In the first step, a common corpus format was developed for the consistent storage of sentence-aligned parallel texts. This encoding scheme is based on XML and focused on the storage of bilingual texts. Conversion software were written in order to handle all different formats that got involved in this project. The number of scripts and tools for small tasks grew rapidly. In the end, the LexEx software bundle comprised over 150 scripts and at least 10 different formats were used for the storage of textual data.

The descriptions above imply the complexity and confusion of data formats and software pieces that were collected in the LexEx study. More and more data was generated by numerous experiments and the collection of results became difficult to handle. The effort on conversion and data handling grew dramatically compared to the work on actual extraction approaches. Furthermore, all approaches to lexical extraction were developed independently and seem to drift away from each other. The necessity of a common platform for the combination of different approaches and the transparent management of textual data was obvious. The system had to support several tasks:

- Data management: All necessary data format have to be supported with a transparent[1] interface. The system has to support standard modes for accessing different sets of data, such as 'write', 'read', 'search' 'add', and 'delete'. Furthermore, tools for conversions between data formats have to be provided. The system has to support the work on large data collections. Access to the data has to be consistent and fast. The data management component has to be extensible with regards to additional data formats and supplementary functions.
- Application management: The system has to handle different applications. Sub-tasks should be defined as modules that are re-usable for various applications. The system has to provide possibilities to change parameter settings and to modify the architecture of the application itself. It has to be extensible and flexible in order to integrate new modules and applications. Processes have to be consistent and robust.
- User interaction: An appropriate user interface has to be integrated in order to provide tools for investigations on test data, intermediate

> data, and final results. Furthermore, the user must have possibilities to control each application by adjusting parameter settings. Corresponding tools for the modification of configurations have to be included. The interface should be easy to use and it should include utilities for different kinds of investigations on textual data.

There are several approaches that focus on the integration of general software modules in the field of language engineering. However, these products tend to be fixed to a certain database format as in the General Architecture for Text Engineering (GATE) (Cunningham et al. 1996) that applies the TIPSTER architecture (Grishman 1997) or as in the CELLAR environment (Simons & Thomson 1995) from the Summer Institute of Linguistic, which applies a specific internal format. Other approaches propose collections of tools for the work with certain encoding standards such as TEI SGML as in the MULTEXT project (Thompson & McKelvie 1996). However, general data architectures often decrease the efficiency of specialized modules. Each sub-task has to be adapted for the usage of the general architecture, which may include a certain overhead that is not needed for this specific task. Therefore, it was decided to develop a new platform for the integration of text processing modules, which supports different data formats that are suitable to specific modules. This toolbox will be referred to as Uplug system in the following.

2. The System and its Components

In this section the architecture of the Uplug system and its components in particular are introduced.

2.1 The Systems Overview

Mainly, the Uplug system is divided into three components. Each component is designed to be extensible and applicable for several purposes. The components are in particular:

- UplugIO - an extensible and transparent I/O interface
- UplugSystem - a launcher for sequences of Uplug modules
- UplugGUI - a graphical user interface for Uplug components

An overview on the system is shown in figure 1. The figure illustrates two integrated applications (Uplug system 1 and Uplug system 2), which are connected to the systems I/O interface and to the graphical user interface. Each application comprises a sequence of modules that perform specific tasks. Furthermore, system 1 includes a loop that iterates the process between module 1 and 3. Each module accesses data collections via the transparent UplugIO component, which is connected to a set of I/O libraries for different data storage formats.

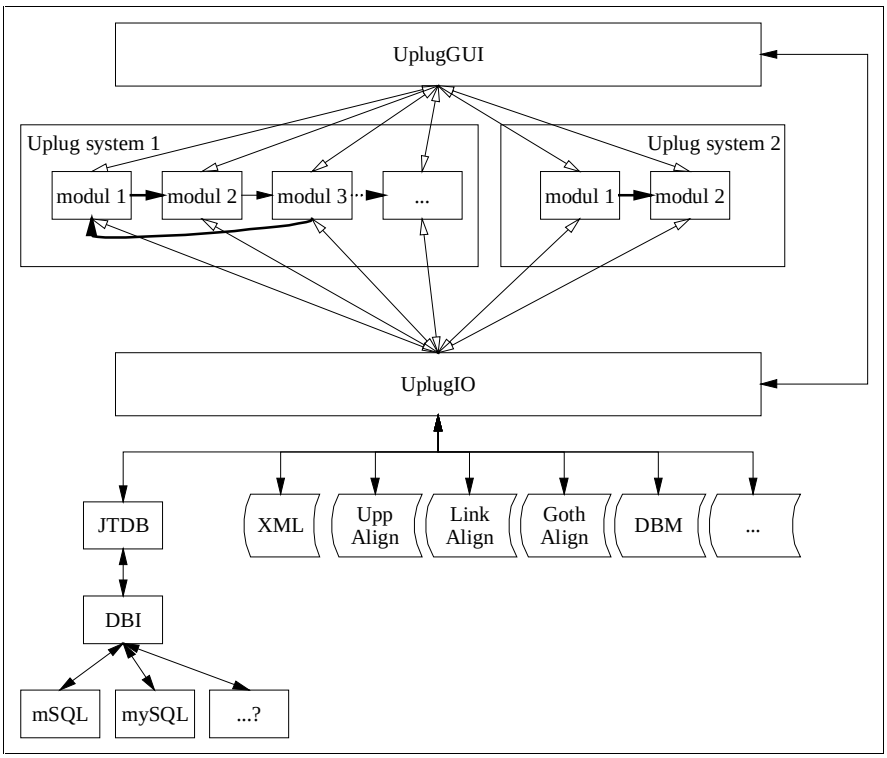

Figure 1. Overview of the Uplug system

Each component will be described in detail in the following sections. However, the general Uplug format for configuration data will be introduced first.

2.2 Configuration Files

All components of the Uplug system apply a general structure for storing system specific parameters (henceforth UplugIni format). This format is supported by special I/O functions, which are integrated in the system. The UplugIni format is readable by humans and straightforward in its structure. Parameter settings are defined in three hierarchies: A parameter **category**, which includes a set of **sub-categories** that contain **feature-structures** in form of pairs of features and their associated values. In this way, classified parameter settings can be defined. Configurations can be extended and modified easily. UplugIni files can be created and modified with common text editors due to the straightforward format. It is possible to add comments and to include external files. Consider the example in figure 1 that illustrates an example of a configuration structure.

```
#-------------------------------------------------
# stream format specifications
#-------------------------------------------------
#include ('/local/uplug/ini/PlugStream.ini')

#-------------------------------------------------
# collections of UplugIO functions
#-------------------------------------------------
[IO libraries]
    general
        file = '(GeneralIO.pl,CollectionIO.pl)'
    corpora
        file = '(UppsalaIO.pl)'
        file = '(LinkoepingIO.pl,GoeteborgIO.pl)'
        file = '(XMLIO.pl)'
    database
        file = '(JTDB.pm)'
#-------------------------------------------------
# define some data streams
#-------------------------------------------------
 [stream specifications]
     sven dictionary
        file = '/local/uplug/data/ensv.dic'
        format = 'DBM'
        DBM type = 'GDBM'
         key = '(source,target)'
         move feature = '(source => temp)'
         move feature = '(target => source)'
         move feature = '(temp => target)'
```

Figure 2. A configuration file in Uplug format

UplugIni files follow a straightforward structure. Each line that starts with the character '#' is considered to be a comment line. However, the special command '#include' forces the system to read the configuration file that is specified within parentheses. Each line that starts with an opening bracket '[' and ends with a closing bracket ']' starts a new *category* section with the name that is specified within the brackets. Other lines that do not include the special character '=' will be interpreted as sub-category names. Features are specified in the current sub-category that is defined in the current category. They start with a name, which has to be unique in the current sub-category and their value is defined in single quotation marks. Valid values include textual data, sequences of textual data or sets of attribute-value pairs. A sequence can be defined on multiple lines with each value enclosed in parentheses. Values in single line sets have to be separated by ','. Both formats may be combined as well (consider the *corpora* feature in figure 1). Sets of attribute value pairs have to be specified on multiple lines with

one pair each. The name of the attribute is separated from its value by the character combination '=>'.

2.3 The UplugIO Component

The UplugIO component comprises a general toolbox for the integration of different data formats and a set of I/O libraries for accessing specific data collections.

2.3.1 General Architecture

The general purpose of this component is to support the access to data collections, which can be processed sequentially i.e. collections, which comprise sets of data with similar structure. Data collections in this sense will be referred to as *data streams*. In general, data streams can be seen as sequences of data records. The I/O component supports general functions for accessing data streams. The following functions are available for each data stream format:

OpenStream
CloseStream
ReadFromStream
WriteToStream
UpdateStreamData
DeleteStreamData
SelectStreamData
SearchData

The functions above require a special parameter for specifying the stream that has to be accessed. Data streams are simply specified by their format and format specific attributes. All necessary specifications are collected in a special data structure, which is used as handle when accessing the data stream. Once specified, the internal structure of the data stream is not significant for the user anymore. All access functions can be applied similarly regardless to the format of the current stream. In this way, the user does not have to deal with internal structures but is provided with a transparent interface for the work on data collections.

The actual methods for accessing each particular stream type are implemented in corresponding I/O libraries. The main component obtains the definitions for each stream format in a special configuration file, *UplugIO.ini*, and calls appropriate functions if a specific stream is accessed. A basic stream format configuration has to include a reference to an appropriate *input* function as well as to an appropriate *output* function. The other parts are replaced by default functions if not otherwise specified. Input/Output functions provide sequential read/write access to the data collection. These basic functions are applied in order to implement functions for searching and updating the stream if no specialized functions are defined for the

stream type. Figure 2 shows a typical stream format specification taken from the UplugIO component.

```
[format specifications]
   plug XML
        open stream function = 'OpenPlugXML'
        input function = 'ReadPlugXML'
        output function = 'WritePlugXML'
        write header function = 'WriteXMLheader'
        write tail function = 'WriteXMLtail'
        select from stream function = 'SelectFromPlugXML'
        functions = '(count => DefaultCount)'
        files = '(file)'
        required stream attributes = '(format)'
        required stream attributes = '(file)'
```

Figure 3. Data stream specifications

2.3.2 Data stream formats

The UplugIO component supports several stream formats. They include specialized formats that are closely related to specific tasks and general formats for general purposes.

Due to the primary application of the Uplug system, the word alignment software, a set of data collections is supported with regard to the data formats that were used in the PLUG project. This includes among others the common XML-based corpus format for bilingual parallel texts and the alignment formats, which are specific to the partners sites. However, the stream accessing tools are designed to be as general as possible and in this way additional stream formats can be included easily.

General data formats that are supported by the Uplug system include interfaces to standard database-management-tools such as SDBM and GDBM, which are common on Unix-alike systems. These interfaces can be used to build simple and fast databases of textual data. The basic database interface is used for several specific applications that require specific data structures. In this way, a corpus annotation stream is implemented, which applies Tipster-alike (Grishman 1998) byte span structures in order to annotate sub-strings in the referred text corpus. Another database interface provides the connection to relational database management systems via the transparent DBI module (Descartes 1997) for Perl. This interface (JTDB) can be seen as an independent database toolbox, which was integrated in the Uplug environment. It provides simple and transparent access to structured data collections plus additional tools for database administration. The

main principle in the JTDB interface is the automatic generation of appropriate SQL queries with regards to a certain internal database structure.

Another feature of the UplugIO component is the possibility of combining data streams into collections. A special stream format (*Collection*) was created to handle sets of data streams. In this way, several data streams can be merged virtually and the user is provided with transparent access to the whole collection similarly to single streams access. The data streams that are included in the collection may be of any format that is supported by the UplugIO component. Furthermore, any combination of stream formats may occur in the collection.

Last to be mentioned here is the possibility of pre-defined data streams. Specifications of data streams can be added in the UplugIO.ini configuration file. Once defined each pre-defined data stream can be referred to by its unique name using the 'stream name' attribute. This is a very convenient way to provide the user with a comprehensible name instead of stream and storage specific attributes. Figure 3 shows a short example of pre-defined data streams from the Uppsala Word Aligner.

```
[stream specifications]
    svenprf
        format = 'plug XML'
        file = '/corpora/PLUG/XML/svenprf.xml'
    svenpeu
        file = '/corpora/PLUG/XML/svenpeu.xml'
        format = 'plug XML'
    svenp[a-z]+
        format = 'Collection'
        stream names = '(svenprf)'
        stream names = '(svenpeu)'
```

Figure 4. Pre-defined data streams

2.4 The UplugSystem Component

The Uplug system intends to combine re-usable modules in order to build special-task applications. Regarding to the idea of the Uplug platform a module can be any external software tool that performs a specific task. In the current stage, the system supports externally executables, perl scripts, and function calls to Perl libraries to be integrated in Uplug applications. Sub-task modules have to be combined in order to build applications of higher complexity. The UplugSystem component provides tools for the construction of complex applications by defining ordered sequences of modules.

```
 [modules]
    tokenize
        command = 'Tokenize'
        configuration = 'Tokenize.ini'
        filename = 'Tokenize.pl'
        type = 'perl lib'
    compile phrases
        command = 'CompilePhrases'
        configuration = 'CompPhrases.ini'
        directory = 'local/uplug/Modules/'
        type = 'perl script'
    extract phrases (source)
        command = 'PhraseExtract'
        configuration = 'SourcePhraseExtract.ini'
        filename = 'PhraseExtract.pl'
        type = 'perl lib'
    segmentation
        command = 'FindLinkSegment'
        configuration = 'Segmentation.ini'
        filename = 'Segmentation.pl'
        type = 'perl lib'
    phrase generator
        command = 'Uplug.pl "phrase generation"'
        directory = '/local/uplug/bin/'
        type = 'perl script'
[systems]
    post-processing
        configdir = './Systems/Scania/sven/Prepare/'
        logfile = 'PreProc.log'
        logfiledir = './log/Scania/sven/'
        modules = '(tokenize)'
        modules = '(phrase generator)'
        modules = '(segmentation)'
        skip modules = '(phrase generator)'
        write logfile = '1'
    phrase generation
        configdir = './Systems/Scania/sven/CompPhrases/'
        logfile = 'CompilePhrases.log'
        logfiledir = './log/Scania/sven/'
        modules = '(compile phrases)'
        modules = '(extract phrases (source))'
        write logfile = '1'
```

Figure 5. Specifications of Uplug systems

Each module has to be defined in the *Uplug.ini* configuration file. The definitions have to follow a certain syntax depending on the module type. A basic Uplug application is defined by a sequence of identifiers that refer to module names in the set of module specifications. Additional parameters can be added such as log-file names and skip definitions for the omission of certain modules. Finally, each

application can be started and the Uplug system is running each module in the sequence starting with the first one in the sequence. The system will stop when the last module is finished. Furthermore, an end module can be specified in order to stop the process at a certain point in the application. Figure 5 shows a sample of a UplugSystem configuration.

The example of a UplugSystem specification in figure 5 presents a simple definition of two Uplug applications, which apply five modules. The first application (*post-processing*) is defined by a sequence of the three modules **tokenise → phrase generator → segmentation**. Parameter settings for each module in the system are stored in the configuration directory (*configdir*) and the log-file (*logfile*) will be created in the log-file directory (*logfiledir*). The second module in this system shows the possibilities of defining sub-systems. The '*phrase generator*' module calls in fact another instance of a Uplug process in order to run the second application, the '*phrase generation*'. The 'phrase generation' system itself includes two modules, which will be run in the sub-system call. In this simple way, hierarchies of Uplug systems can be defined easily.

As mentioned earlier, a module can be mainly any kind of executable or script that can be run on the system. However, each module should fit in the application it is part of. The Uplug system is designed to be as general as possible and therefore no restrictions were defined for the integration of additional modules. The compilation of applications is up to the user. Consistency of interactions between modules is not guaranteed by the system. Each module can be completely independent from any Uplug component. The most basic Uplug application is simply a batch process of different programs. However, integrated modules may use Uplug components for the interaction with each other. Modules runs separately. Interactions are defined by means of data that are produced in each step. Interactions between modules can be supported via the UplugIO component and a straightforward configuration structure. Each module in the Uplug environment may use a simple parameter file, which is structured as follows:

1. Configurations are stored by means of UplugIni structures.
2. The configuration file includes
    - an 'input' category for the specification of input data streams
    - an 'output' category for the specification of output data streams
    - a 'parameter' category for the specification of module specific parameters

In Figure 6 a simple example of such module configuration file is shown.

UplugIni files are very convenient for storing parameter settings for single modules. The structure is compatible with the UplugIO component. The data

stream specifications can be applied directly in order to access corresponding data collections. Pre-defined streams can be used and parameters can be ordered in hierarchies. Each sub-category name in the input/output sections defines a unique identifier for each stream within the current application. Data stream specifications are taken from previous specifications in the sequence of modules if they refer to the same identifier. In this way, standard configurations can be defined for specific modules that can be applied for different Uplug applications.

```
[input]
    corpus
        stream name = 'corpus (stem forms)'
[output]
    source token frequencies
        stream name = 'source token frequencies'
    target token frequencies
        stream name = 'target token frequencies'
[parameter]
    lower case
        source = '1'
        target = '1'
    runtime
        print progress = '1'
    stemmer
        source = 'sv'
        target = 'en'
    token
        delimiter = '  '
        grep = 'contains_alphabetic'
```

Figure 6. An example of a module configuration file

The configuration format is also supported by the graphical user interface UplugGUI. Using UplugIni files, each data stream can be inspected and parameters can be set via the interface.

The UplugSystem component supports iterative processing. Each system may include a loop in the sequence of modules. Loops are defined by specifying the index of the start module (*loop start*), the index of the end module (*loop end*), and the number of iterations (*loop iterations*) that have to be carried out. Further interior loops can be added by defining appropriate sub-systems.

2.5 The UplugGUI Component

The Uplug system provides a graphical user interface for the work with Uplug applications. This interface is window and mouse oriented based on Perl/Tk scripts. It comprises various tools for the construction, configuration, and

application of Uplug systems. The actual appearance of the interface depends on information in corresponding configuration files. It can be adjusted by modifying appropriate parameters. In this way, the main menu can be modified, pre-defined streams can be set, and parameter types and options can be defined. The main window displays the sequence of modules of the current application. The UplugGUI uses information from configuration files if they exist for a specific module. It provides convenient tools for the adjustment of parameters. For this, standard Tk widgets are used to set parameter values according to the type and widget specifications that are defined in the UplugGUI configuration. The widget type can be specified in the module configuration file as well. Each parameter can be associated with a certain type in the '*widgets*' category. Figure 6 illustrates the widget specification from a typical module configuration file.

```
 [widgets]
    runtime
        print progress = 'checkbox'
    token
        delimiter = 'entry'
        grep = 'optionmenu
(numeric,alphabetic,contains_alphabetic)'
        minimal length = 'scale (1,10,1,1)'
    token pair
        maximal distance = 'scale (1,20,1,1)'
        minimal frequency = 'scale (1,10,1,1)'
        minimal length difference = 'scale (1)'
```

Figure 7. Specifications of parameter widgets in module configuration files

The current version of the UplugGUI supports four different widgets:

- Entry widgets for simple textual data
- Checkboxes for boolean flags
- Option-menus for selecting values from a certain set of options
- Scales for numeric parameters with valid values in a certain range

The UplugGUI creates widgets according to the specification that were found. The default type is the entry field, which will be created for each parameter that is not otherwise specified. Furthermore, command buttons can be added to specific attributes as well. In this way, additional tools can be used for setting specific parameters such as file dialogs for setting name and location of file parameters. Figure 8 shows two screen shots of such parameter configuration dialogues.

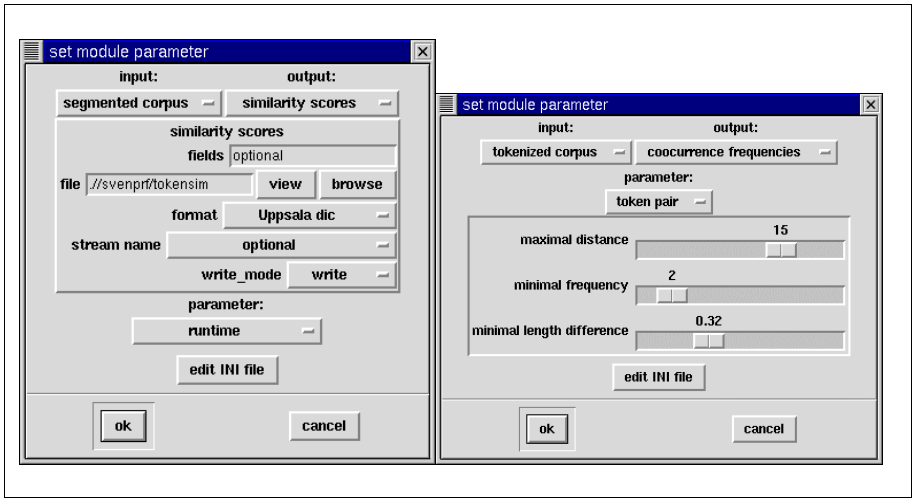

Figure 8. Setting parameters with the UplugGUI

Another feature of the UplugGUI is the possibility of inspecting data streams. Each stream, which is defined in a module configuration file, can be inspected by simply clicking on corresponding buttons in the main window. Furthermore, it is possible to open any Uplug data stream from the interface by specifying corresponding attributes. The system creates data windows that list sequentially read data records. In the data window tools are provided to query the collection and to store data records in different formats at new locations. In this way, intermediate results from each module can be inspected even if the application is still in progress. Data collections can be converted easily and prepared for specific investigations.

Finally, Uplug applications can be certainly started from the graphical interface. Depending on the configuration the process will be started using a system shell. In multi-task environments, the process may run in the background. In this way, several applications may be started simultaneously from the user interface. However, interactions between different processes have to be considered. The system does not support any consistency checks so far.

3. Example Applications

The primary application of the Uplug system so far is the Uppsala Word Alignment (UWA). In this application, bilingual parallel texts are processed in order to identify translation equivalents in parallel texts. The UWA can be used to mark all correspondences that could be identified in the text or to extract bilingual lexicons from the text corpus. The system assumes sentence aligned text corpora

and runs through a sequence of modules and sub-systems. A special focus in the design of the UWA was set on modularity. The word alignment process may combine different sequences of sub-task modules. Figure 9 illustrates the main architecture of the UWA. It represents one possible combination of modules for a word alignment application. Each module performs a specific task. They can be removed, substituted by new modules, and additional modules may be included in order to create modified alignment applications.

1. pre-processing
    - tokenization
    - generation of multi-word collocations (source & target language)
    - text segmentation (identification of multi-word units in the text)
2. investigations on string similarity
    - the longest common subsequence ratio (LCSR)
    - weighted LCSR
3. stemming (reducing words to stem forms)
4. co-occurrence statistics
    - frequencies counts (token frequencies, co-occurrence frequencies)
    - calculation of co-occurrence statistics (Dice, Mutual information, t-score)
    - investigations on low frequent pairs
5. word and multi-word alignment
6. automatic filtering
7. compilation of a bilingual dictionary
8. iteration: continue with module 3

Figure 9. The UWA system

Although the UWA is the main application of the Uplug system, additional systems were developed on the same platform. They apply different components and carry out various tasks related to text corpus processing. Among them, parts of the UWA are used as separate systems. Uplug applications include systems for

- phrase generation (based on the compilation of contiguous collocations)
- text segmentation (including tokenization and identification of multi-word-units)
- alignment evaluation based on a gold standard
- bilingual concordances (applying the UplugIO component)
- the work with lexical databases (applying UplugIO)

4. Conclusions

The Uplug-system including its main application, the Uppsala Word Aligner, represents Uppsala's contribution to the common word alignment system, which is currently under development. The purpose with this software is to provide a modular platform for the integration of text processing tools. Special attention is given to the development of a general system that supports further extensions. The current version of the Uplug-system, however, is intended for processing bilingual texts from the project's corpus. A special focus was set on the integration of different storage formats. The system is supposed to support access to different data formats instead of creating a new internal structure that has to be used by each application. In this way, existing storage standards can be applied by corresponding applications and data conversions can be avoided. Specific data formats are often optimised for certain tasks. Keeping the same format increases the time efficiency of sub-tasks. The system is designed to be very general and it allows comprehensive modifications in its configuration. No explicit restrictions were defined for the integration of modules. Due to this fact, the consistency in Uplug applications is very much up to the user that created certain applications. The system is still under development and a prototype is currently applied for specific tasks. In future, additional applications and further data formats will be integrated in the system.

The Uplug system represents an applicable platform for extensive investigations on textual data. It provides a general and extensible architecture with an integrated user interface and a multi-purpose data interface. Its main application is focused on the extraction of information from large text collections, in particular multi-lingual parallel texts.

Notes

1 Transparency in this sense means the invisibility of internal structures for the user.

References

Ahrenberg, L., M. Merkel, K. Mühlenbock, D. Ridings, A. Sågvall Hein and J. Tiedemann (1998), 'Parallel corpora in Linköping, Uppsala and Göteborg'. Project application, available at http://stp.ling.uu.se/ ~corpora/plug/.

Cunningham, H., Y. Wilks, R. J. Gaizauskas (1996), 'Software Infrastructure for Language Engineering', in: *Proceedings of the AISB Workshop on Language Engineering for Document Analysis and Recognition*, University of Sussex.

Descartes, A. (1997), 'DBI: The Database Interface'. *The Perl Journal*, Issue 5.

Grishman, R. (1998), 'TIPSTER Text Architecture Design, Version 3.1', New York University, available at http://www.itl.gov/iaui/894.02 /related_projects/tipster/download.htm

Simons, G. F. and J. V. Thomson (1995), 'Multilingual data processing in the CELLAR environment', in: *Linguistic Databases*, University of Groningen, Centre for Language and Cognition and Centre of Behavioural and Cognitive Neurosciences

Tiedemann, J. (1997), 'Automatical lexicon extraction from aligned bilingual corpora'. Diploma thesis, Otto-von-Guericke-University, Magdeburg, Department of Computer Science.

Tiedemann, J. (1998), 'Extraction of translation equivalents from parallel corpora', in: *Proceedings of the 11th Nordic Conference on Computational Linguistics*, Copenhagen 28-29 January 1998 (NODALIDA'98), Center for Sprogteknologi, University of Copenhagen. 120–128.

Tjong Kim Sang, E. (1996a), 'Converting the Scania Framemaker documents to TEI SGML'. Technical report, Department of Linguistics, Uppsala University.

Tjong Kim Sang, E. (1996b), 'Aligning the Scania corpus'. Technical report, Department of Linguistics, Uppsala University.

Thompson, H and D. McKelvie. (1996), 'A software architecture for simple, efficient SGML applications', in: *Proceedings of SGML Europe '96*, Munich.