How to Integrate *SProUT's* XTDL Grammar Interpreter in Other Applications

Jakub Piskorski
German Research Center for Artificial Intelligence
Stuhsatzenhausweg 3, 66123 Saarbrücken, Germany
piskorsk@dfki.de

VERSION APRIL 2005
(SPROUT VERSION 4.0 AND HIGHER)

1 Introduction

SProUT provides functionality that allows for integration of its XTDL grammar interpreter or a cascade of such grammar interpreters in other applications or frameworks. In this documentation, we briefly describe the steps which have to be undertaken in order to make use of this feature. We focus here on creating single interpreter instances, whereas cascaded system architectures are described elsewhere.

The rest of this paper is structured as follows. Firstly, in section 2, we explain the syntax of the configuration file for the *XTDL* interpreter. Subsequently in section 3, we show some sample code fragments which could potentially be used by developers in the process of integrating and deploying *SProUT* in other applications. The Apendix A contains documentation of some relevant Java classes.

Note, that the first task in the process of integrating *SProUT* in other applications is to provide all indispensable linguistic resources. For this purpose one can use the already existing resources used in the projects created with *SproUT*s Integrated Development Environment. Alternatively, all indispensable resources can be created as follows. Tokenizer and gazetteer resources are created with the regular compiler [1]. Resources for the extended gazetteer, morphology, and sentence boundary recognizer are created via an application of dedicated ANT scripts [2], dedicated compilation scripts or a call to adequate methods in the corresponding Java classes as described in [3,4,5]. Finally XTDL grammars are created with the *SProUT* IDE. The new type of resources (*SProUT* version 3.10.2 and higher), including

the rule prioritization file and the sentence-boundary limited rule file are created as well with the IDE. Please refer to documentations in the reference list for further details.

IMPORTANT: Probably most problems users encounter while using *SProUT* have to do with invalid or incompatible type hierarchies and inconsistent linguistic resources. Therefore, please take an insight into the definition of the type hierarchy you are currently using before trying to localize the problems elsewhere.

2 Configuration File

In order to run the XTDL interpreter, a configuration file must be prepared which is then used for initialization[1]. The same effect can be achieved by using a corresponding property object (see class `java.util.Properties`). We will now list all properties which have to be defined in the configuration file. Some of them are obligatory, whereas others depending on the context may be skipped. All obligatory properties will be marked by a preceding asterisk.

2.1 Input Text Configuration

`InputFile` **-** Specifies the path to the input file to be analyzed. Note that this property is not obligatory. However if you do not specify it, then reading a new input text can be performed directly in the code by a corresponding call to one of the methods: `setNewInputText()` or `setNewInputTextfromString()` in the class `de.dfki.lt.sprout.runtime.interpreter.SproutInterpreter`, once the interpreter has been initialized.

Example: `InputFile=D:/SPROUT/sample.txt`

Please note, that from now on there is an option of specifying additional input data, e.g., a file containing XML representation of a list of typed feature structures (generated eventually by another SProUT instance or external processing components), an array of typed feature structures, or a 'dom' object. These options are available via a call to certain methods (listed below) in the `de.dfki.lt.sprout.runtime.interpreter.SproutInterpreter` class.

```
SetNewXMLInputText()
setNewXMLInputTextfromString()
setNewInputArray()
setNewInputSource()
```

For the details please see the documentation at the end of this document.

[1] The IDE provides (Tools → Generate Install package / RuntimeConfig → Generate RuntimeConfig) an option of automatically generating such configuration files. If You create the configuration file in this manner, You can directly proceed to section 3.

`* CharSet` – Specifies the character set used for reading the input text. For German, English, French, Spanish, and Italian set it to `ISO-8859-1`. For Polish and Czech use `ISO-8859-2`, for Chinese use `GB2312`, and for Japanese use `euc-jp`. Please refer for further details to `java.nio.charset.Charset` in the java API documentation available at `http://www.java.sun.com`.[2]

Example: `CharSet=ISO-8859-1`

2.2 Logging Configuration

`LoggerPropertyConfiguration` – Specifies the path to the configuration file for the logger (SProUT uses log4j software – http://logging.apache.org/log4j/docs/). If this property is missing or the path is invalid, then default logging to standard output will be activated. IMPORTANT: Please note, that if system variable ***log4j.configuration*** is defined, then all other logging settings have no effect.

2.3 Type Hierarchy Configuration

`* TypeHierarchy` Specifies the path to the file containing the type hierarchy.

Example: `TypeHierarchy=D:/SPROUT/sampleHierarchy.grm`

2.4 Components Configuration

Depending on what processing resources will be used (and configured), a list of corresponding properties must be set to `true` (active) or `false` (not active). If a certain processing resource will not be used, then the corresponding property does not necessarily have to be specified in the configuration file (it will be set by default to `false`). Currently, there are seven processing resources: tokenizer, gazetteer, extended gazetteer, morphology, external morphology, sentence boundary recognition, and external input converter. The corresponding properties are: `Tokenizer`, `Gazetter`, `ExtendedGazetteer`, `Morphology`, `ExternalMorphology`, `SentenceBoundaryRecognizer`, `ExternalInputConverter`. The external input converter simply converts external data (e.g., XML File, DOM-object, ArrayList of TFSs) into an input data stream for the XTDL interpreter.

Note that, in case the tokenizer is not active, then all other processing resources except XML Input converter will be automatically deactivated (since they all rely on tokenization

[2] In order to list all character sets for which support is available in the current Java virtual machine, take advantage of the static method `java.nio.charset.Charset.available()`.

information). In previous system versions the tokenizer was always used. This is not anymore the case since there is an option of using XML files as input. Please note also that for Chinese and Japanese tokenization is performed by the external morphology component. Further, it is not allowed to set both Morphology and External Morphology to `true` since only one morphology component may be used at a time.

Example:
```
Tokenizer=true
Gazetteer=true
ExtendeGazetteer=true
Morphology=true
SentenceBoundaryRecognizer=true
ExternalInputConverter=false
```

2.5 Morphology Configuration

`Lexicon` Specifies the path to the lexicon file.

Example: `Lexicon=D:/SPROUT/sampleLexicon.slx`

Currently, resources for English, German, Dutch, French, Italian, and Spanish are provided.

2.6 External Morphology Configuration

Currently four external morphological components have been integrated for Chinese (*Shanxi*), Japanese (*Chasen*), Czech and Polish (*Morfeusz*). They run as servers or are called on demand as subprograms.[3] In order to configure an external morphology component, up to five properties must be defined. This depends upon the language. We list these properties beneath.

`MorphologyLanguage` – Specifies the language of the external morphology (choose one of the values: `chinese`, `japanese`, `polish`, or `czech`)

`MorphologyPort` – Specifies the communication port (does not apply to Chinese).

`MorphologyServerAdress` – Specifies the address of the server on which external morphology component is running.

[3] We assume that the users are familiar with the issues concerning starting the corresponding server applications (and programs). Please note, that in case of Polish morphology component, the system first tries to find a corresponding dynamic library (usually delivered with the system) which allows for accessing the component directly (no client/server modus). If this library can not be found, then the system tries to establish a connection to the corresponding server, as specified in the configuration file. The latter modus is somewhat less efficient in terms of processing speed (if you work under Linux, this is the only possibility to work in a client/server mode).

MorphologyApplicationPath - Specifies the path to the executable application file.

MorphologyDisambiguation - Enforces the morphology component to perform disambiguation. Note that currently this option is only applicable for Czech morphology component.

The following example illustrates how to configure the Czech morphology component, which runs on a local machine.

Example:
```
MorphologyLanguage=czech
MorphologyPort=0
MorphologyServerAdress=localhost
MorphologyDisambiguation=true
```

The table beneath gives an overview which properties must be specified for particular language

Property\Language	Chinese	Japanese	Polish	Czech
MorphologyLanguage	*	*	*	*
MorphologyPort				*
Morphology ServerAddress	*		*	*
MorphologyDisambiguation				*
MorphologyApplicationFile		*		

2.7 Tokenizer Configuration

In order to configure the tokenizer, several resources are needed.

MainTokenClassesFSM - Defines the path to the file containing the main token classes.

SeparatorClassesFSM - Defines the path to the file containing the separators.

WSClassesFSM - Defines the path to the file containing the whitespaces.

SubClassesFSM - Defines the path to the file containing token subclasses.

ScheduleFile - Defines the path to the file containing the schedule.

MainClassNameList - Defines the path to the file with main token class names

SubClassNameList - Defines the path to the file with token subclasses names

UnknownType - Defines the name for the 'unknown' token class (tokens are qualified as unknown in case they do not match any of the defined main token classes). Note, that this definition must be compatible with the one in the type hierarchy, otherwise further processing will be stopped.

InitialFinalSeparatorTrimming - Specifies (**true**/**false**) whether the special separator trimming of (potential) initial and final separators will be performed.[4] By default this property is set to **true**.

Example:
```
MainTokenClassesFSM=D:/SPROUT/MainTokenClasses.fsm
SeparatorClassesFSM=D:/SPROUT/SeparatorClasses.fsm
WSClassesFSM= D:/SPROUT /WhiteSpaces.fsm
SubClassesFSM= D:/SPROUT /SpecificClasses.fsm
ScheduleFile=D:/SPROUT/Schedule.txt
MainClassNameList=D:/SPROUT/MainTokenClassesNames.txt
SubClassNameList=D:/SPROUT/SpecificClassesNames.txt
UnknownType=other
InitialFinalSeparatorTrimming=true
```

2.8 Sentence Boundary Recognition

In order to configure the sentence boundary recognizer, the following properties can be used.

SentenceBoundaryRecognizerResources - Defines the path to the file with the resources needed by the sentence boundary recognizer.

SentenceBoundaryLimitedRules - Defines the path to the file containing the list of rules which are sentence-boundary sensitive. If this property is not specified, then all rules ignore sentence boundaries.

UseTwoEndOfLineRule - Specifies (**true**/**false**) whether the two-end-of-line rule is active or not (by default it is deactivated). In case it has been activated, then all empty lines enforce an introduction of a sentence boundary.

[4] For the sake of clarity, we give an example here. Lets consider the string 'http://www.dfki.de.' By default it will be segmented by the tokenizer into 10 tokens, namely: 'http' ':' '/' '/' 'www' '.' 'dfki' '.' 'de' '.' The trailing period is responsible for such a segmentation since the whole string does not belong to any particular token class ('unknown' type). If we remove the trailing period, then the string would be classified as a single token of type 'email address'. If we activate the option of trimming initial/final separators, then all tokens classified as 'unknown' are segmented as follows: firstly, initial/final separators are isolated (in our example we have only the final separator), and secondly the tokenizer tries to classify the remaining part. This would lead in our case to a segmentation of the input string into 'http://www.dfki.de' and '.' which is intuitively exactly what we wanted to achieve. Using this option turned to come in particular in handy for token segmentation/classification near sentence boundaries.

Example:
```
SentenceBoundaryRecognizerResources=D:/SPROUT/SB.ssb
SentenceBoundaryLimitedRules=D:/SPROUT/SB_sensitive_rules.txt
UseTwoEndOfLineRule=true
```

2.9 Gazetteer Configuration

Four properties must be defined for the gazetteer.

`MainClassesFSM` – Defines a path to the file with the compiled gazetteer.

`MainClassNames` – Defines a path to the file with gazetteer class names.

`CaseInsensitive` – Specifies if the gazetteer works in a case insensitive mode (**`true`**) or not (**`false`**).

`SeparateWords` – Specifies whether a space separates gazetteer words. Set to **`true`** for all languages but Japanese and Chinese.

The following example illustrates a configuration of a gazetteer for an Indoeuropean language in a case sensitive mode.

Example:
```
MainClassesFSM=D:/SPROUT/GazetteerSample.fsm
MainClassNames=D:/SPROUT/GazetteerSampleNames.txt
CaseInsensitive=false
SeparateWords=true
```

2.10 Extended Gazetteer Configuration

Two properties must be defined for the extended gazetteer.

`ExtGazetteerResource` – Defines a path to the file with the compiled gazetteer resources.

`ExtGazCaseInsensitive` – Specifies whether the extended gazetteer works in a case insensitive mode (`true`) or not (`false`). 'Case insensitive' mode means that the input text is downcased before gazetteer look-up is performed. However, switching between the two modes does not effect the entries in the gazetteer.

Example:
```
ExtGazetteerResource=D:/SPROUT/GazetteerSample.sgz
ExtGazCaseInsensitive=false
```

2.11 XTDL Grammar Configuration

`* XTDLGrammar` – Specifies the path to the file containing the input grammar.

Example: `XTDLGrammar=D:/SPROUT/sampleXTDLGrammar.fsm`

2.12 Grammar Interpreter Configuration

There are several options how the grammar interpreter can be configured. This includes: output structure generation, output structure merging, partial coreference resolution, and rule prioritization.

2.12.1 Output Configuration

`GenerateStartEndPositions` – Specifies whether the output structures generated by the interpreter include start/end position information (token and character positions) of the matched text spans. Note that the default value of this attribute is set to **`false`**. Choosing the **`true`** value will be penalized with respect to processing speed.

`OutputType` – Specifies the type of the structures which will be generated in the output stream. There are three possible values for this property: `OUTPUT_INTERPRETER_RESULT` (default option: output only structures produced by the interpreter), `OUTPUT_ALL` (output structures produced by the interpreter and all structures produced by other processing components), `OUTPUT_INTERPRETER_RESULTS_AND_UNCONSUMED_ITEMS` (output structures produced by the interpreter and all unconsumed items produced by other processing components).

`OutputFilterTypes` – Defines the list of types (separated by semicolon) for further filtering of the structures in the output stream, i.e., if a candidate structure in the output stream is not subsumed by any of the specified types, then such structure will be deleted from the output stream[5].

Advanced output configuration options:

`NormalizeFS` – Specifies whether **`(true/false)`** sets in output structures will be flattened. Default value is set to **`false`**.

`SimplifyFS` – Specifies whether **`(true/false)`** duplicate elements (equivalent) in sets in output structures will be removed. Default value is set to **`false`**.

[5] This option is not yet available

Here is an example of output configuration.

Example:
```
GenerateStartEndPositions=true
OutputType=OUTPUT_INTERPRETER_RESULT
OutputFilterTypes=ne-organization;ne-person
```

The defined properties has the following effect: output only structures generated by the interpreter, and only those which are of type **ne-organization** or **ne-person**.

2.12.2 Output Structure Merging

SProUTs grammar interpreter might potentially return multiple results for a single text span. Such ambiguities might result from different reasons (e.g., processing components return more than one interpretation for certain text span, several paths in the grammar might lead to the same result, etc.). In order to reduce the number of output structures generated by the grammar interpreter, some merging operations are provided.

EqualityMerge - Specifies whether (**true**/**false**) duplicate structures should be removed. By default this property is set to **false**.

MergingOption - Specifies which merging option will be used for reducing the number of the output structures. There are four possible values for this property: `MERGE_NOT_ACTIVE` (merging is not activated), `MERGE_MOST_SPECIFIC` (the most specific output structure will be returned), `MERGE_MOST_GENERAL` (the most general output structure will be returned), `MERGE_UNIFICATION` (the output structures are maximally integrated via a sequence of unification operations, which resembles in some way traditional template merging). The default value of this property is `MERGE_NOT_ACTIVE`.

MergingStructure - Specifies the scope of the output structure to which merging operations will be applied. There are two options: (a) only the final OUT-structures are merged - `MERGE_OUT_STRUCTURE`, (b) whole output structures returned by the interpreter are considered - `MERGE_WHOLE_STRUCTURE`. The default option is `MERGE_OUT_STRUCTURE`.

Example:
```
EqualityMerge=true
MergingOption=MERGE_MOST_SPECIFIC
```

2.12.3 Coreference Resolution

A partial coreference resolution tool may be activated on demand. It takes as input the output structures generated by the grammar interpreter so far, potentially containing user-defined information on variants of the recognized entities of certain type, and performs an additional pass through the input text, in order to discover mentions of previously recognized entities.

The variant specification is done explicitly by defining additional attributes, e.g., VARIANT, on the RHS of grammar rules, which contain a list of all variant forms[6] (e.g., obtained by concatenating some of the constituents of the full name). In order to activate and use the coreference resolution tool following properties must be defined.

ApplyCoreferencer - Specifies whether (**true**/**false**) coreference resolution mechanism is active or not. By default this property is set to **false**.

CoreferencerRelevantAttributes - Defines the list of attributes (separated by semicolon) which are taken into account by the coreference resolution tool in order to obtain information on variants.

CoreferencerFrameActive - Specifies whether (**true**/**false**) the context frame is active or not. By default this property is set to **false**.

CoreferencerLeftBoundary - Defines the left-most boundary of the context frame (in tokens).

CoreferencerRightBoundary - Defines the right-most boundary of the context frame (in tokens).

If the value of the left-most and right-most boundary of the context frame are set to *x* and *y* respectively, it means that a candidate mention must refer to an entity which appears within the last *x* tokens from the token position of the candidate or within the next *y* tokens from the token position of the candidate.

Example:

```
ApplyCoreferencer=true
CoreferencerRelevantAttributes=PERSON_VARIANT;ORG_ABBREV
CoreferencerFrameActive=true
CoreferencerLeftBoundary=100
CoreferencerRightBoundary=100
```

2.12.4 Rule prioritization

SProUTs' mechanism for rule prioritization can be deployed in order to further reduce the number of generated output structures. The ruleset is divided into two sets, one with neutral rules (which are not considered in the process of rule prioritization) and a second set of linearly ordered rules. If any of the rules from the second set match a given text fragment,

[6] Currently, the value of such attributes has to be of type 'string' and different variants should be separated with '|' symbol (e.g., "Kowalski | Prof. Kowalski"). In the future versions of SProUT there will be more flexibility.

then only the one with the highest priority will be triggered for generating a corresponding output structure.

RulePrioritization - Specifies whether (**true**/**false**) the rule prioritization mechanism is activated or deactivated.

RulePrioritizationFile - Defines the path to the file containing the prioritized rules.

Example:
```
RulePrioritization=true
RulePrioritizationFile=D:/SPROUT/prioritized_rules.txt
```

2.13 Using system variables and user-defined properties

It is allowed to use system variables and user-defined properties for defining Sprout relevant properties, which simplify the definition of the configuration file. In order to access the value of a system variable or previously defined property `XYZ` the following syntax has to be used: `${XYZ}`. We illustrate the usage of such macros for defining root paths in an example. Let's say, we want to configure the gazetteer, and the root path should be set to '`D:/SPROUT/gazetteer_resources`'. The example in section 2.6 should be modified as follows.

```
GazResourcePath=D:/SPROUT/gazetteer_resources

MainClassesFSM=${GazResourcePath}/GazetteerSample.fsm
MainClassNames=${GazResourcePath}/GazetteerSampleNames.txt
CaseInsensitive=false
SeparateWords=true
```

Alternatively if `GazResourcePath` is defined as a system variable, the first line of our example should be removed. IMPORTANT: Notice that, in `${…}` constructions system variables override user-definable properties.

2.14 Final Example

Finally, we give an example of a complete configuration file. Let us assume that one wants to use tokenizer, extended gazetteer, sentence boundary recognizer (with the two-end-of-line rule activated) and the morphology component. Further, we assume that all indispensable resources are saved in the directory `D:/SPROUT/RESOURCES`. Finally, we would like to use the following options of the grammar interpreter:

- output only structures generated by the interpreter, including the start/end positions of the matched text fragments in these output structures,
- rule prioritization,
- using the output structure merging option which results in generating only the most specific output structures (consider only the OUT-structures while merging)
- performing partial coreference resolution (relevant attribute is VARIANT) with no contextual frame (the scope is then the whole document).

A following configuration file could be used.

```
# Input Text Settings
InputFile=D:/SPROUT/sample.txt
CharSet=ISO-8859-1

# Define a root directory for resources
Resources=D:/SPROUT/RESOURCES

# Type Hierarchy Settings
TypeHierarchy=${Resources}/sampleHierarchy.grm

# Active Component Settings
Tokenizer=true
ExtendeGazetteer=true
Morphology=true
SentenceBoundaryRecognizer=true

# Morphology Settings
Lexicon=${Resources}/sampleLexicon.slx

# Tokenizer Settings
MainTokenClassesFSM=${Resources}/MainTokenClasses.fsm
SeparatorClassesFSM=${Resources}/SeparatorClasses.fsm
WSClassesFSM=${Resources}/WhiteSpaces.fsm
SubClassesFSM=${Resources}/SpecificClasses.fsm
ScheduleFile=${Resources}/Schedule.txt
MainClassNameList=${Resources}/MainTokenClassesNames.txt
SubClassNameList=${Resources}/SpecificClassesNames.txt
UnknownType=other

# Gazetteer Settings
ExtGazetteerResource=${Resources}/GazetteerSample.sgz
ExtGazCaseInsensitive=false

# Sentence Boundary Recognizer Settings
SentenceBoundaryRecognizerResources=${Resources}/SB.ssb
SentenceBoundaryLimitedRules=${Resources}/SB_sensitive_rules.txt
UseTwoEndOfLineRule=true

# Grammar Settings
XTDLGrammar=${Resources}/sampleXTDLGrammar.fsm

# ---- Grammar Interpreter Settings ----
# Output Configuration
GenerateStartEndPositions=true
```

```
OutputType=OUTPUT_INTERPRETER_RESULTS

# Output Structure Merging
MergingOption=MERGE_MOST_SPECIFIC
MergingStructure=MERGE_OUT_STRUCTURE

# Coreference Resoultion
ApplyCoreferencer=true
CoreferencerRelevantAttributes=VARIANT

# Rule Prioritization
RulePrioritization=true
RulePrioritizationFile=${Resources}/prioritized_rules.txt

# ---- Grammar Interpreter Settings ----
```

3 Sample Code

In this section, we present sample code which can be used as a guideline for integrating *SProUT's* interpreter. Below we give a simplified example of a function which takes as argument the name of the configuration file and the name of the input file. It analyzes the input file, and sends the result to the standard output. In order to build an instance of the XTDL Interpreter the class `de.dfki.lt.sprout.runtime.interpreter.SproutInterpreter` must be used. Note that the interpreter returns an array list, where each element of the resulting array list is again an array list containing all interpretations found for a given text fragment. Single interpretations are represented as objects of type `de.dfki.lt.sprout.runtime.MatchInfo`.

```
import java.util.*;
import de.dfki.lt.sprout.runtime.*;
import de.dfki.lt.sprout.runtime.interpreter.*;
import org.apache.log4j.*;
...
public void runInterpreter(String ConfigurationFile, String InputFile)
 { // An ArrayList object for storing the output returned by the interpreter
   ArrayList Output = null;

   // Create new object representing an interpreter
   SproutInterpreter IT = new SproutInterpreter ();

   // Initialize the interpreter with a configuration file
   boolean status = IT.initialize(ConfigurationFile);

   // Set the path to an input text disregarding the configuration file
   IT.setNewInputText(InputFile);

   // Run the interpreter if configuration was successfull
   if(status==true) Output = IT.RunInterpreter();
   else System.out.println("System could not be initialized properly !!!");

   // Iterate over a list of matches produced by the interpreter
   if(Output!=null)
    { for(int i=0;i<Output.size();i++)
       { System.out.println("Match no. " + i);
         System.out.println("-------------------------");
         System.out.println(" ");
         // get disjunction of applicable rules for i-th match
```

```
          ArrayList vMI = (ArrayList)Output.get(i);
          // iterate over the disjunction
           for(int j=0;j<vMI.size();j++)
            { System.out.println("Result: " + j);
              MatchInfo Result = (MatchInfo)vMI.get(j);
              // write the rule name
              System.out.println("Rule Name: " + Result.getRuleName());
              // write positional information
              System.out.println("iCStart: " + Result.getCStart() + ", iCEnd: " +
               Result.getCEnd() + ", iStart: " + Result.getStart() + ", iEnd: " +
               Result.getEnd());
              // write the full rule TFS
               System.out.println("Structure: ");
               Result.getFs().printFS();
              System.out.println(" ");
            }
        }
    }
 }
```

Note that it is also possible to initialize the interpreter directly with a `java.util.Properties` object. One would need the following block of code

```
import java.util.*;
...
Properties InterpreterProperties = new Properties();
try { FileInputStream FIS = new FileInputStream(ConfigurationFile);
      InterpreterProperties.load(FIS);
      FIS.close();
    } catch (IOException e) { System.out.println("IO ERROR"); }
boolean status = IT.initialize(InterpreterProperties);
```

Furthermore, the interpreter can be applied to a set of input texts. For this purpose one should use `de.dfki.lt.sprout.runtime.interpreter.SproutInterpreter.RunInterpreter(String FileList) method`. Here is a sample code, which could be deployed for this purpose.

```
...
String FileList = "..."; // file containing a list of files to be analyzed
ArrayList [] MultipleOutput = null;

if(status==true) MultipleOutput = IT.RunInterpreter(FileList);
else System.out.println("System could not be initialized properly !!!");

// iterate over a list of matches produced by interpreter
if(MultipleOutput!=null)
 { for(int k = 0;k<MultipleOutput.length;k++)
    { System.out.println("Output for file no. " + k);
      System.out.println(" ");
      if(MultipleOutput[k]!=null)
       { for(int i=0;i<MultipleOutput[k].size ();i++)
           { System.out.println("Match no. " + i);
             System.out.println("-------------------------");
             System.out.println(" ");
             // get disjunction of applicable rules for i-th match
              ArrayList vMI = (ArrayList)MultipleOutput[k].get (i);
```

```
            // iterate over the disjunction
            for(int j=0;j<vMI.size();j++)
              { System.out.println("Result: " + j);
                MatchInfo Result = (MatchInfo)vMI.get(j);
                // write the rule name
                System.out.println("Rule Name: " + Result.getRuleName());
                // write positional information
                System.out.println("iCStart: " + Result.getCStart() + ", iCEnd: " +
                  Result.getCEnd() + ", iStart: " + Result.getStart() + ", iEnd: "
                  + Result.getEnd());
                // write the full rule TFS
                 System.out.println("Structure: ");
                 Result.getFs().printFS();
                System.out.println(" ");
              }
            }
          }
      }
  }
```

Some additional information concerning the current settings of the interpreter can be obtained by a call to one of the following public methods of the class **de.dfki.lt.sprout.runtime.SproutInterpreter**:

```
public String getInputTextFile();
public ShUG getTypeHierarchy();
public String getMorphologyLanguage();
public String getCharSet();
public String getActiveComponents();
```

Actually, the example we started this section with has been implemented as the class **de.dfki.lt.sprout.runtime.interpreter.SproutBatch**. In order to run it use the following syntax:

MSWindows version:

```
java -Xmx640M -cp %1 de.dfki.lt.sprout.runtime.interpreter.SproutBatch %2
[%3]
```

Linux Version:

```
java -Xmx640M -cp %1 de.dfki.lt.sprout.runtime.interpreter.SproutBatch %2 [%3]
```

where %1 ($1) specifies the path to: (a) the directory containing *SProUT* Java archive (**sprout.jar** or **sprout.zip**) and/or a path to the root directory containing the compiled **SproutBatch** class, and (b) the log4j-1.2.8.jar archive[7] (used for logging since version 3.10.2). The second parameter %2 ($2) is a path to the XTDL interpreter configuration file,

[7] For internal stuff: The latest versions of all jar archives which are used by *SProUT* can be found in DFKIs internal directory /project/cl/sprout/test/resource/lib

and %3 ($3) is an optional argument with the name of the input file. If the last parameter is used, then the definition of the input file in the configuration file will be overridden.

The class `de.dfki.lt.sprout.runtime.interpreter.SproutManagaer` gives You an opportunity to instantiate several SProUT systems at once and run them interchangeably in a multi-threading secure way on various input data.

For further details concerning the provided functionality, please see the Java code documentation in Appendix A.

4 Acknowledgements

Providing the new functionality for integrating SProUTs interpreter in other applications presented in this paper could not have been possible without a help of Witold Drożdżyński to whom I am thankful. I am also grateful to Hans-Ulrich Krieger for reviewing this documentation and additional hints.

5 References

[1] W. Drożdżyński, J. Piskorski. *Tokenization in SProUT*. SProUT developers API – http://sprout.dfki.de/documentations/api

[2] U. Schaefer, D. Beck. *Ant documentation for SProUT*. URL: http://sprout.dfki.de/documentations/ant

[3] J. Piskorski. *Lexicon Look-up component in SProUT*. SProUT developers API – http://sprout.dfki.de/documentations/api

[4] J. Piskorski. *Extended Gazetteer in SProUT*. SProUT developers API – http://sprout.dfki.de/documentations/api

[5] J. Piskorski. *Sentence Boundary Recognition in SProUT*. SProUT developers API – http://sprout.dfki.de/documentations/api

APPENDIX A - JAVA DOCUMENTATION

JAVA DOCUMENTATION FOR THE CLASS SproutInterpreter

de.dfki.lt.sprout.runtime.interpreter
Class SproutInterpreter

java.lang.Object
└ **de.dfki.lt.sprout.runtime.interpreter.SproutInterpreter**

---

public class **SproutInterpreter**
extends java.lang.Object

This class allows you to run and test the SProUT Interpreter as a stand-alone application. The resources must be created as described in the SProUT documentation. In particular, use the GUI version of SProUT to generate the necessary grammar files. There are two ways of initializing an object of this class: by configuring a configuration script, or by creating a corresonding 'Properties' object. All features and functionalities of the grammar interpreter and all processing resources available in the 3.10.2 version of the SProUT GUI are also available via using this class.

Constructor Summary

SproutInterpreter()	Simple constructor.
SproutInterpreter(java.lang.String FileName)	Constructor which takes the name of the configuration file as argument

Method Summary

static boolean	**configureLogger**(java.lang.String LoggerPropertyFile) Configure the logger according to the Logger property file.
java.lang.String	**getActiveComponents**() Returns a list of active processing components
java.lang.String	**getCharSet**() Returns character set being used

ExtendedGazetteer	**getExtendedGazetteer**() Returns the current extended gazetteer object
Gazetteer	**getGazetteer**() Returns the current gazetteer object
java.lang.String	**getID**() Gets the ID of the Sprout Interpreter
java.util.ArrayList	**getInputArray**() This method returns the input array object - for storing additional input data.
java.lang.String	**getInputText**() This method returns the pointer to the current input text
java.lang.String	**getInputTextFile**() This method returns the file name of the current input text
Morphology	**getMorphology**() Returns the current morphology object
java.lang.String	**getMorphologyLanguage**() Returns the current morphology language
SentenceBoundaryRecognizer	**getSystemSentenceBoundaryRecognizer**() Returns the current sentence bouindary object
Tokenizer	**getTokenizer**() Returns the current tokenizer object
ShUG	**getTypeHierarchy**() Returns the current type hierarchy used by the interpreter.
java.lang.String	**getXTDLGrammar**() Returns the name of the current input grammar
boolean	**initialize**(java.util.Properties CFG) Initializes a 'SproutInterpreter' object according to a 'Properties' object passed as argument.
boolean	**initialize**(java.util.Properties CFG, ShUG currentTypeHierarchy, java.lang.String LoggerConfigurationFile) Initializes a 'SproutInterpreter' object according to a 'Properties' object passed as argument, and eventually an input type hierarchy which overrides the corresponding 'TypeHierarchy' setting in the 'Properties' object, and additionally a path to the logger configuration file which overrides also the corresponding setting 'LoggerPropertyConfiguration' in 'Properties' object.

boolean	**initialize**(java.lang.String FileName) Initializes an 'SproutInterpreter' object according to a configuration script passed as argument
boolean	**initialize**(java.lang.String FileName, ShUG currentTypeHierarchy, java.lang.String LoggerConfigurationFile) Initializes an 'SproutInterpreter' object according to a configuration script passed as argument, and eventually an input type hierarchy which overrides the corresponding 'TypeHierarchy' setting in the configuration script, and additionally a path to the logger configuration file which overrides also the corresponding setting 'LoggerPropertyConfiguration' in the Sprout interpreter configuration file.
java.util.ArrayList	**RunInterpreter**() This method allows for applying the interpreter to the current input data.
Java.util.ArrayList[]	**RunInterpreter**(java.lang.String InputFileList) This method allows for applying the interpreter to a set of input texts.
java.util.ArrayList	**RunInterpreterForLargeData**() This method allows for applying the interpreter to the current input data.
static void	**setBasicLogger**() Activates the default basic logger (standard output).
void	**setCharSet**(java.lang.String CharSet) Set character set
void	**setID**(java.lang.String name) Sets the ID of the Sprout Interpreter
void	**setNewInputArray**(java.util.ArrayList IA) This method selects a new input data from an input array.
void	**setNewInputSource**(org.xml.sax.InputSource SourceXML) This method selects a new input data from an InputSource object
boolean	**setNewInputText**(java.lang.String FileName) This method selects a new input text via loading an input file.
void	**setNewInputTextfromString**(java.lang.String Input) This method selects a new input text from a string.
boolean	**setNewXMLInputText**(java.lang.String FileName) This method selects a new input text via loading an xml input file.

void	setNewXMLInputTextfromString(java.lang.String Input) This method selects a new xml input text from a string.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

SproutInterpreter

public **SproutInterpreter**()
Simple constructor.

SproutInterpreter

public **SproutInterpreter**(java.lang.String FileName)
Constructor which takes the name of the configuration file as argument

Parameters:

`FileName` - the name of the configuration file

Method Detail

setID

public void **setID**(java.lang.String name)
Sets the ID of the Sprout Interpreter

Parameters:

`name` - ID of the Sprout instance Note: if the input argument is `null`, the ID will not be changed

---

getID

public java.lang.String **getID**()
Gets the ID of the Sprout Interpreter

Returns:

the ID of the Sprout instance

---

setNewInputText

public boolean **setNewInputText**(java.lang.String FileName)

This method selects a new input text via loading an input file.
Parameters:
`FileName` - the name of the new input file
Returns:
`true` if this operation was succesfull, i.e., the specified file exists, or `false` otherwise. Note that if this operation is not successfull, then a pointer to a current input text is preserved !!!

---

setNewXMLInputText

public boolean **setNewXMLInputText**(java.lang.String FileName)
This method selects a new input text via loading an xml input file.
Parameters:
`FileName` - the name of the new xml input file
Returns:
`true` if this operation was succesfull, i.e., the specified file exists, or `false` otherwise. Note that if this operation is not successfull, then a pointer to a current input text is preserved !!!

---

setNewInputTextfromString

public void **setNewInputTextfromString**(java.lang.String Input)
This method selects a new input text from a string.
Parameters:
`Input` - the string object containing the new input text

---

setNewXMLInputTextfromString

public void **setNewXMLInputTextfromString**(java.lang.String Input)
This method selects a new xml input text from a string.
Parameters:
`Input` - the string object containing the new xml input text

---

setNewInputArray

public void **setNewInputArray**(java.util.ArrayList IA)
This method selects a new input data from an input array.
Parameters:
`IA` - an ArrayList object containing the input data

---

setNewInputSource

public void **setNewInputSource**(org.xml.sax.InputSource SourceXML)

This method selects a new input data from an InputSource object

Parameters:

`SourceXML` - object containing the input data

---

getInputTextFile

public java.lang.String **getInputTextFile**()

This method returns the file name of the current input text

---

getInputArray

public java.util.ArrayList **getInputArray**()

This method returns the input array object - for storing additional input data.

---

getInputText

public java.lang.String **getInputText**()

This method returns the pointer to the current input text

---

getTypeHierarchy

public ShUG **getTypeHierarchy**()

Returns the current type hierarchy used by the interpreter.

---

getMorphologyLanguage

public java.lang.String **getMorphologyLanguage**()

Returns the current morphology language

---

getMorphology

public Morphology **getMorphology**()

Returns the current morphology object

---

getSystemSentenceBoundaryRecognizer

public SentenceBoundaryRecognizer **getSystemSentenceBoundaryRecognizer**()

Returns the current sentence bouindary object

getTokenizer

public Tokenizer **getTokenizer**()
Returns the current tokenizer object

getCharSet

public java.lang.String **getCharSet**()
Returns character set being used

setCharSet

public void **setCharSet**(java.lang.String CharSet)
Set character set
Parameters:
`CharSet` - character set

getXTDLGrammar

public java.lang.String **getXTDLGrammar**()
Returns the name of the current input grammar

getGazetteer

public Gazetteer **getGazetteer**()
Returns the current gazetteer object

getExtendedGazetteer

public ExtendedGazetteer **getExtendedGazetteer**()
Returns the current extended gazetteer object

getActiveComponents

public java.lang.String **getActiveComponents**()
Returns a list of active processing components

initialize

public boolean **initialize**(java.lang.String FileName)

Initializes an 'SproutInterpreter' object according to a configuration script passed as argument

Parameters:

`FileName` - the name of the configuration file

Returns:

`true` if this operation was succesfull, or `false` otherwise.

---

initialize

public boolean **initialize**(java.lang.String FileName,
ShUG currentTypeHierarchy,
java.lang.String LoggerConfigurationFile)

Initializes an 'SproutInterpreter' object according to a configuration script passed as argument, and eventually an input type hierarchy which overrides the corresponding 'TypeHierarchy' setting in the configuration script, and additionally a path to the logger configuration file which overrides also the corresponding setting 'LoggerPropertyConfiguration' in the Sprout interpreter configuration file. In order not to override the 'TypeHierarchy' or 'LoggerPropertyConfiguration' settings in the configuration script please set the last two arguments to `null`.

Parameters:

`FileName` - the name of the configuration file

`currentTypeHierarchy` - SHuG object representing the input type hierarchy

`LoggerConfigurationFile` - the path to the logger configuration file

Returns:

`true` if this operation was succesfull, or `false` otherwise.

---

configureLogger

public static boolean **configureLogger**(java.lang.String LoggerPropertyFile)

Configure the logger according to the Logger property file. Please note, that in case the logger has already been configured the application of this method has no effect.

Parameters:

`LoggerPropertyFile` - the property file used for configuring the logger

Returns:

`true` if the configuration was successfull, or `false` otherwise.

---

setBasicLogger

public static void **setBasicLogger**()

Activates the default basic logger (standard output).

---

initialize

public boolean **initialize**(java.util.Properties CFG)

Initializes a 'SproutInterpreter' object according to a 'Properties' object passed as argument.

Returns:

`true` if this operation was succesfull, or `false` otherwise.

---

initialize

public boolean **initialize**(java.util.Properties CFG,
ShUG currentTypeHierarchy,
java.lang.String LoggerConfigurationFile)

Initializes a 'SproutInterpreter' object according to a 'Properties' object passed as argument, and eventually an input type hierarchy which overrides the corresponding 'TypeHierarchy' setting in the 'Properties' object, and additionally a path to the logger configuration file which overrides also the corresponding setting 'LoggerPropertyConfiguration' in 'Properties' object. In order not to override the 'TypeHierarchy' or 'LoggerPropertyConfiguration' settings in the configuration script please set the last two arguments to `null`.

Parameters:

`CFG` - Properties object representing the configuration
`currentTypeHierarchy` - SHuG object representing the input type hierarchy
`LoggerConfigurationFile` - the path to the logger configuration file

Returns:

`true` if this operation was succesfull, or `false` otherwise.

RunInterpreterForLargeData

public java.util.ArrayList **RunInterpreterForLargeData**()

This method allows for applying the interpreter to the current input data. It returns an object containing a list of disjunctions of TFS, where each such disjunction represents the rules which were applicable at a given position.

Returns:

an ArrayList object including the results generated by the interpreter. Note that each element of the resulting array list is again an array list containing of MatchInfo objects (representing single interpretations)

See Also:

`de.dfki.lt.sprout.runtime.MatchInfo`

RunInterpreter

public java.util.ArrayList **RunInterpreter**()

This method allows for applying the interpreter to the current input data. It returns an object containing a list of disjunctions of TFS, where each such disjunction represents the rules which were applicable at a given position.

Returns:

an ArrayList object including the results generated by the interpreter. Note that each element of the resulting array list is again an array list containing of MatchInfo objects (representing single interpretations)

See Also:

`de.dfki.lt.sprout.runtime.MatchInfo`

RunInterpreter

public java.util.ArrayList[] **RunInterpreter**(java.lang.String InputFileList)

This method allows for applying the interpreter to a set of input texts. It returns an array of objects representing the matches for each of the input texts (each such object contains a list of disjunctions of TFS, where a single disjunction represents the rules which were applicable at a given position).

Parameters:

`InputFileList` - a file containing the pathes to the files with input texts

Returns:

an array of ArrayList objects which represent the results returned for the consecutive input texts (ArrayList[i] -> results for document i). Note that the elements of each single ArrayList object are again ArrayList objects which represent the interpretations (MatchInfo objects) for consecutive text spans.

See Also:

`de.dfki.lt.sprout.runtime.MatchInfo`

JAVA DOCUMENTATION FOR THE CLASS SproutManager

---

de.dfki.lt.sprout.runtime.interpreter

Class SproutManager

java.lang.Object
  └ **de.dfki.lt.sprout.runtime.interpreter.SproutManager**

---

public class **SproutManager**
extends java.lang.Object

This class implements a SProUT manager which allows to manage several SProUT instances and apply them on demand to different input data. For initialization

purposes a list of SProUT configuration files is required. Please note that this class is a singleton, and the initialization process is performed automatically and only once. For initialization do the following: Define a system variable called SPROUT_MANAGER which specifies the path to the configuration file of the Sprout manager. This file should include a list of paths to SProUT configuration files (each line contains a single path). Additionally, another system variable SPROUT_CONFIGURATION_ROOT can be defined which points to the root directory where SProUT configuration files are located. If the latter variable is not defined, the paths in the configuration file of the SProUT manager must be absolute. Example: Configuration file for the manager is located in D:/Sprout/SM.cfg. It includes several paths to SProUT configuration files, whereas the root directory for these files is located at D:/Sprout/Configurations. The system variables should be defined as below SPROUT_MANAGER=D:/Sprout/SM.cfg SPROUT_CONFIGURATION_ROOT=D:/Sprout/Configurations

---

Method Summary	
static java.util.ArrayList	**ApplySproutToInputArray**(java.util.ArrayList input, int sproutID) This method allows for applying a given SProUT instance to the input data.
static java.util.ArrayList	**ApplySproutToInputSource**(org.xml.sax.InputSource input, int sproutID) This method allows for applying a given SProUT instance to the input data.
static java.util.ArrayList	**ApplySproutToInputText**(java.lang.String input, int sproutID) This method allows for applying a given SProUT instance to the input data.
static java.util.ArrayList	**ApplySproutToInputTextFromString**(java.lang.String input, int sproutID) This method allows for applying a given SProUT instance to the input data.
static java.util.ArrayList	**ApplySproutToXMLInputFromString**(java.lang.String input, int sproutID) This method allows for applying a given SProUT instance to the input data.
static java.util.ArrayList	**ApplySproutToXMLInputText**(java.lang.String input, int sproutID) This method allows for applying a given SProUT instance to the input data.
static java.lang.StringBuffer	**convertToXML**(FeatureStructure F) Converts Feature structures to XML (synchronized version)

int	**getNumSproutInstances**() Retruns the number of SProUT instances in the SProUT manager.
SproutInterpreter	**getSproutInterpreter**(int i) Returns i-th SProUT instance
java.lang.String	**getSproutInterpreterConfigurationFile**(int i) Returns the path to the configuration file of the i-th SProUT instance
static SproutManager	**getSproutManager**() Returns the only one SproutManager object

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Method Detail

getNumSproutInstances

public int **getNumSproutInstances**()

Retruns the number of SProUT instances in the SProUT manager.

---

getSproutInterpreter

public SproutInterpreter **getSproutInterpreter**(int i)

Returns i-th SProUT instance

Parameters:

i - index of the SProUT instance

Returns:

i-th instance of SProUT instance (@see de.dfki.lt.sprout.runtime.interpreter.SproutInterpreter) or null if the index is invalid.

---

getSproutInterpreterConfigurationFile

public java.lang.String **getSproutInterpreterConfigurationFile**(int i)

Returns the path to the configuration file of the i-th SProUT instance

Parameters:

i - index of the SProUT instance

Returns:

path to the configuration file of the i-th SProUT instance or null if the index is invalid.

---

getSproutManager

public static SproutManager **getSproutManager**()
Returns the only one SproutManager object

---

convertToXML

public static java.lang.StringBuffer **convertToXML**(FeatureStructure F)
Converts Feature structures to XML (synchronized version)
Parameters:
`F` - input feature structure
Returns:
StringBuffer representation of XML version of the input structure or `null` if the input feature structure points to `null`

ApplySproutToInputArray

public static java.util.ArrayList **ApplySproutToInputArray**(java.util.ArrayList input,
int sproutID)
This method allows for applying a given SProUT instance to the input data. It returns an object containing a list of disjunctions of TFS, where each such disjunction represents the rules which were applicable at a given position.
Parameters:
`input` - an input ArrayList containing the input data (list of feature structures)
`sproutID` - index of the SProUT instance to be applied
Returns:
an ArrayList object including the results generated by the interpreter. Note that each element of the resulting array list is again an array list containing of MatchInfo objects (representing single interpretations)
See Also:
`de.dfki.lt.sprout.runtime.MatchInfo`

ApplySproutToInputSource

public static java.util.ArrayList **ApplySproutToInputSource**(org.xml.sax.InputSource input,
int sproutID)
This method allows for applying a given SProUT instance to the input data. It returns an object containing a list of disjunctions of TFS, where each such disjunction represents the rules which were applicable at a given position.
Parameters:
`input` - an InputSource object containing the input data (@see org.xml.sax.InputSource)
`sproutID` - index of the SProUT instance to be applied
Returns:

an ArrayList object including the results generated by the interpreter. Note that each element of the resulting array list is again an array list containing of MatchInfo objects (representing single interpretations)

See Also:

`de.dfki.lt.sprout.runtime.MatchInfo`

ApplySproutToInputText

public static java.util.ArrayList **ApplySproutToInputText**(java.lang.String input, int sproutID)

This method allows for applying a given SProUT instance to the input data. It returns an object containing a list of disjunctions of TFS, where each such disjunction represents the rules which were applicable at a given position.

Parameters:

`input` - a path to the file containing the input text

`sproutID` - index of the SProUT instance to be applied

Returns:

an ArrayList object including the results generated by the interpreter. Note that each element of the resulting array list is again an array list containing of MatchInfo objects (representing single interpretations)

See Also:

`de.dfki.lt.sprout.runtime.MatchInfo`

ApplySproutToInputTextFromString

public static java.util.ArrayList **ApplySproutToInputTextFromString**(java.lang.String input, int sproutID)

This method allows for applying a given SProUT instance to the input data. It returns an object containing a list of disjunctions of TFS, where each such disjunction represents the rules which were applicable at a given position.

Parameters:

`input` - a string containing the input text

`sproutID` - index of the SProUT instance to be applied

Returns:

an ArrayList object including the results generated by the interpreter. Note that each element of the resulting array list is again an array list containing of MatchInfo objects (representing single interpretations)

See Also:

`de.dfki.lt.sprout.runtime.MatchInfo`

ApplySproutToXMLInputText

public static java.util.ArrayList **ApplySproutToXMLInputText**(java.lang.String input,
int sproutID)

This method allows for applying a given SProUT instance to the input data. It returns an object containing a list of disjunctions of TFS, where each such disjunction represents the rules which were applicable at a given position.

Parameters:

`input` - a path to the file containing the input data in XML format

`sproutID` - index of the SProUT instance to be applied

Returns:

an ArrayList object including the results generated by the interpreter. Note that each element of the resulting array list is again an array list containing of MatchInfo objects (representing single interpretations)

See Also:

`de.dfki.lt.sprout.runtime.MatchInfo`

ApplySproutToXMLInputFromString

public static java.util.ArrayList **ApplySproutToXMLInputFromString**(java.lang.String input,
int sproutID)

This method allows for applying a given SProUT instance to the input data. It returns an object containing a list of disjunctions of TFS, where each such disjunction represents the rules which were applicable at a given position.

Parameters:

`input` - a string containing the input data in XML format

`sproutID` - index of the SProUT instance to be applied

Returns:

an ArrayList object including the results generated by the interpreter. Note that each element of the resulting array list is again an array list containing of MatchInfo objects (representing single interpretations)

See Also:

`de.dfki.lt.sprout.runtime.MatchInfo`
