JTaCo & SProUTomat: Automatic Evaluation and Testing of Multilingual Language Technology Resources and Components

Christian Bering[†], **Ulrich Schäfer**[*]

[†]Computational Linguistics Department, Saarland University
P.O.Box 151150, D-66041 Saarbrücken, Germany
christian.bering@acrolinx.com
[*]Language Technology Lab, German Research Center for Artificial Intelligence (DFKI) GmbH
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
ulrich.schaefer@dfki.de

Abstract

We describe JTaCo, a tool for automatic evaluation of language technology components against annotated corpora, and SProUTomat, a tool for building, testing and evaluating a complex general-purpose multilingual natural language text processor including its linguistic resources (lingware). The JTaCo tool can be used to define mappings between the markup of an annotated corpus and the markup produced by the natural language processor to be evaluated. JTaCo also generates detailed statistics and reports that help the user to inspect errors in the NLP output. SProUTomat embeds a batch version of JTaCo and runs it after compiling the complex NLP system and its multilingual resources. The resources are developed, maintained and extended in a distributed manner by multiple authors and projects, i.e., the source code stored in a version control system is modified frequently. The aim of JTaCo & SProUTomat is to warrant a high level of quality and overall stability of the system and its lingware.

1. Introduction

The development of multilingual resources for language technology components is a tedious and error-prone task. Resources (lingware) like morphologies, lexica, grammars, gazetteers, etc. for multiple languages can only be developed in a distributed manner, i.e., many people work on different resources.

However, the resulting systems are supposed to deliver the same good recognition quality for each language. Dependencies of resources and subsystems may lead to suboptimal performance, e.g., reduced recognition rates, of the overall systems in case of errors creeping in during the development process. Hence, in analogy to software engineering, testing and evaluation of the developed lingware has to be carried out on a regular basis, both for quality assurance and comparability of results in different languages. Annotated natural language corpora can be thought of as providing a rich and potentially very useful body of test material in this context. However, it is often not possible to flexibly incorporate the material at hand into the development process. The reasons are manifold: Not only may annotations in different sources be of very diverse nature, but the NLP component under development usually generates a markup in yet another format defined by the development environment.

In this paper, we describe a framework consisting of two major components, JTaCo and SProUTomat, that facilitates frequent (e.g., daily) building, testing and evaluation of multilingual language components and resources in a quality assurance and development cycle as depicted in Figure 1. We have implemented and will demonstrate the framework for the multilingual SProUT processor. However, the concepts and mechanisms described could be applied to any other resource-intensive natural language processing system.

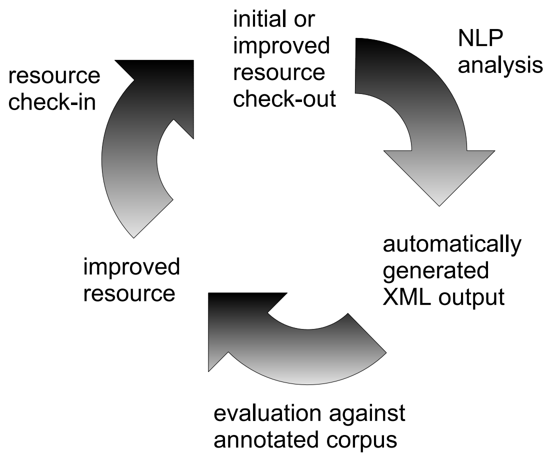

initial or improved resource check-out
NLP analysis
automatically generated XML output
evaluation against annotated corpus
improved resource
resource check-in

Figure 1: Quality assurance and development cycle for multilingual linguistic resources.

2. SProUT

SProUT is a shallow, multilingual, general-purpose natural language processor (Drozdzynski et al., 2004). SProUT comes with a powerful, declarative grammar formalism XTDL that combines finite-state techniques and typed feature structures with structure sharing and a fully-fledged, efficiently encoded type hierarchy—in contrast to systems like GATE (Cunningham et al., 2002) that support only simple attribute-value pairs.

SProUT rules consist of regular expressions over typed feature structures[1]. A rule is matched against a sequence of input feature structures which are filled by basic components like tokenisers, morphology or gazetteer lookup run-

[1]The acronym SProUT stands for Shallow Processing with Unification and Typed feature structures. SProUT's homepage is `http://sprout.dfki.de`.

ning on input text or, in more complex cases, XML output from external NLP components or even output from previous SProUT grammar stages.
The matching condition is unifiability of the input sequence with the expanded regular expression of the left hand side of a rule. In case of a match, feature structure unification is used to transport information from the matching left hand side to the output feature structure on the right hand side of the rule. The output feature structure can then, e.g., be transformed to any XML format.
The SProUT system provides basic components like tokenisers, morphologies and domain-specific gazetteers for languages such as English, German, French, Spanish, Greek, Japanese, Italian, Chinese, Polish and Czech, and comes with a user-friendly integrated development environment (IDE). The current main applications of SProUT are information extraction and named entity recognition (NER).
To illustrate the SProUT formalism, we give a short example in Figure 2 of a grammar rule that recognises river names. The rule matches either expressions consisting of an (unknown) capitalised word (via token type match), followed by a noun with stem *river* or *brook* (via the English morphology component; disjunction has a higher precedence than concatenation), or Gazetteer entries of type *gaz_river* containing English river names represented by the Gazetteer type *gaz_river*. The generated output structure of type *ne-location* contains a location type *river* and the location name transported via the coreference symbol $\boxed{\text{loc\_name}}$. To sum up, this rule recognises both unknown river names (via a pattern involving morphology lookup that tolerates morphologic variants) and known river names (via a gazetteer match), using a concise, declarative pattern and returning a structured description.
SProUT has been and is currently used in many research and industrial projects for opinion and text mining, information extraction, automatic hyperlinking, question answering and semantic web applications (Drozdzynski et al., 2004).

3. JTaCo

The aim of JTaCo (Bering, 2004; Bering et al., 2003) is to allow the developer of an NLP component or resource, e.g., of a grammar, to make unified use of variably annotated source material for testing. The component developer provides suitably, i.e., usually semi-manually or manually marked-up reference sources on the one hand, and a parser or similar NLP component on the other hand. JTaCo extracts the original annotation from the corpus, compares this annotation with the markup the component in question generates for the same input, and generates statistics and reports from the comparison results[2].
Since a focus of JTaCo lies on the integration of diverse manual annotation schemes one the one hand and differing NLP components on the other, JTaCo employs a very modular architecture in which its different processing stages allow independent adaptations to varying input and different environments. JTaCo is realised as a pluggable lightweight, mostly architecture-independent framework. Currently, there are two JTaCo plug-in realisations for usage with grammars developed in SProUT: A GUI plug-in integrated into the SProUT IDE, and a batch version integrated into SProUTomat.

[2]JTaCo stands for Java Tagging Comparator.

3.1. JTaCo's Processing Stages

JTaCo works in four separate transformational processing stages. Figure 3 gives an overview of these stages, of their input and the results they generate. The process starts from an annotated written corpus against which the NLP component or resource is to be tested. In the first step, JTaCo uses an *AnnotationParser* to separate the corpus into

- the 'raw' text contained in the corpus (i.e., the text without any annotation) and
- its *true* annotation (interchangeably also called the *reference* or *manual* annotation).

The extracted text is fed into the *Parser* (or a similar component) which the developer wants to test, yielding the annotation to compare with the manual annotation. The comparison is executed by a *TaggingComparator*. The comparator's result in turn is used by an *OutputGenerator* to select, format and output the needed information.

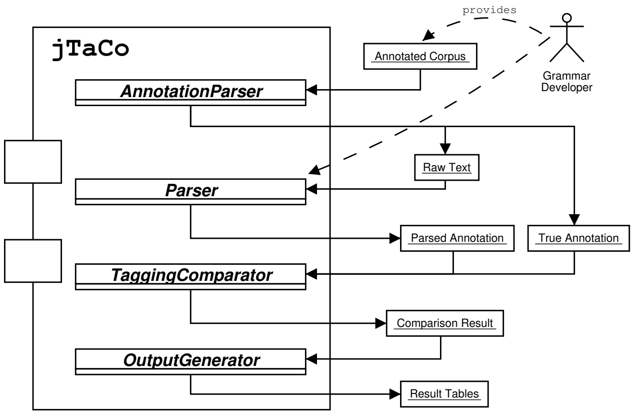


Figure 3: An overview of JTaCo's processing stages and the (intermediate) results they yield.

There are two main advantages gained from such a modular architecture: On the one hand, the abstract representations in the intermediate results hide details specific to the corpus or component used. For instance, differing types of annotations are mapped to an abstract annotation representation, for which a comparison operation – i.e., especially the notion of equality between entities in the two annotations – can be defined in an adequately flexible manner, and the underlying annotated sources as well as NLP components can be exchanged transparently. Thus, whenever a new annotation format or component makes it necessary to integrate a tailored module into JTaCo, the capabilities of the new module can readily interact with existing functionality of other modules.
The second, more practically relevant advantage is that the settings of any one stage can be changed, and the process at that stage rerun with the new settings without having to

$$
\begin{gathered}
\text{river} :> \left(\left(\begin{bmatrix} \textit{token} & \\ \text{TYPE} & \textit{first\_capital\_word} \\ \text{SURFACE} & \boxed{\text{loc\_name}} \end{bmatrix} \bullet \left(\begin{bmatrix} \textit{morph} & \\ \text{STEM} & \texttt{"river"} \\ \text{POS} & \textit{noun} \\ \text{SURFACE} & \boxed{\text{key}} \end{bmatrix} \mid \begin{bmatrix} \textit{morph} & \\ \text{STEM} & \texttt{"brook"} \\ \text{POS} & \textit{noun} \\ \text{SURFACE} & \boxed{\text{key}} \end{bmatrix} \right) \right) \right) \\
\mid \begin{bmatrix} \textit{gazetteer} & \\ \text{GTYPE} & \textit{gaz\_river} \\ \text{CONCEPT} & \boxed{\text{loc\_name}} \\ \text{DESIGNATOR} & \boxed{\text{key}} \end{bmatrix} \rightarrow \begin{bmatrix} \textit{ne-location} & \\ \text{LOCTYPE} & \textit{river} \\ \text{LOCNAME} & \boxed{\text{loc\_name}} \\ \text{DESCRIPTOR} & \boxed{\text{key}} \end{bmatrix} .
\end{gathered}
$$

Figure 2: A SProUT grammar rule recognizing river names. Boxed feature values denote structure sharing, type names are typeset in italics. The dot after the first token indicates concatenation, the vertical bars separate alternatives.

re-iterate the previous process stages, as long as their results are still available. This can be especially useful for the last two stages in an interactive environment (i.e., comparison and report generation), where the developer might want to experiment with different settings without repeatedly having to rerun the probably time-consuming processes of reading the corpus and parsing it.

For each of the stages, a JTaCo plug-in uses one or more processing realisations adapted to the desired representations. In what follows, we will draw upon the implementations integrated into SProUT and SProUTomat to illustrate the information flow in JTaCo.

3.2. Reading the Annotated Corpus

For use in the following processing stages, JTaCo extracts from the annotated corpus the 'raw' content, i.e., the written text without any markup, on the one hand, and the reference annotation on the other. Both the extraction of the text and of the annotation can be configured according to the specific annotation scheme. E.g., a corpus usually not only contains the annotated textual material, but also meta-information intended for, e.g., administrative purposes. Such information has to be exluded from the text extracted to be used for testing. Currently, JTaCo includes support for annotations which satisfy certain regular constraints and for XML annotations such as found in MUC corpora (Grishman and Sundheim, 1996). For use with SProUT, JTaCo transforms the XML-encoded entities into typed feature structures.

As an illustration, consider the following MUC time expression:

```
<TIMEX TYPE="DATE">07-21-96</TIMEX>
```

The textual content consists just of the date expression *07-21-96*. JTaCo transforms the tag information as well as the surface and character offsets into feature-value pairs in a feature structure:

$$
\begin{bmatrix} \textit{timex} & \\ \text{TYPE} & \texttt{"DATE"} \\ \text{CSTART} & \texttt{"27"} \\ \text{CEND} & \texttt{"34"} \\ \text{SURFACE} & \texttt{"07-21-96"} \end{bmatrix}
$$

Here, CSTART and CEND indicate the inclusive start and end character positions of the annotated element in the 'raw' text, i.e., without counting the markup. The resulting reference annotation is the collection of all feature structures generated from the corpus. More complex, embedded annotations would be translated in a similar manner.

3.3. Parsing the Extracted Text

In this second processing stage, JTaCo feeds the NLP component which the developer wants to test with the text retrieved from the previous stage, and the NLP component in turn produces some specific markup of the text. As in the previous stage, JTaCo transforms this annotation into a format which it can compare with the reference annotation. For the previously employed example expression, *07-21-96*, SProUT's named entity recognition markup delivers structured output in an XML-encoded typed feature structure[3], where CSTART and CEND indicate start and end character positions of the matched named entity in the input text:

$$
\begin{bmatrix} \textit{point} & \\ \text{SPEC} & \textit{temp-point} \\ \text{MUC-TYPE} & \textit{date} \\ \text{CSTART} & \texttt{"27"} \\ \text{CEND} & \texttt{"34"} \\ \text{SURFACE} & \texttt{"07-21-96"} \\ \text{YEAR} & \texttt{"1996"} \\ \text{MONTH} & \texttt{"07"} \\ \text{DOFM} & \texttt{"21"} \end{bmatrix}
$$

3.4. Comparing the Annotations

In this stage, the annotations obtained from the two previous tranformation processes are compared, i.e., the 'manual' annotation read directly from the corpus, and the 'parsed' annotation obtained through the NLP component. For JTaCo, an annotation is a collection of tags, where a tag consists of some linguistic information about a piece of text. Minimally, a tag contains

- some name, e.g., a linguistic label,

[3]Transformation of typed feature structures and general XML markup is discussed in the context of the upcoming ISO standard in (Lee et al., 2004). Actually, SProUT's default XML output format is very close to the proposed ISO format for typed feature structures.

- the surface string to which the label applies,
- token count information about where this string is found in the corpus.

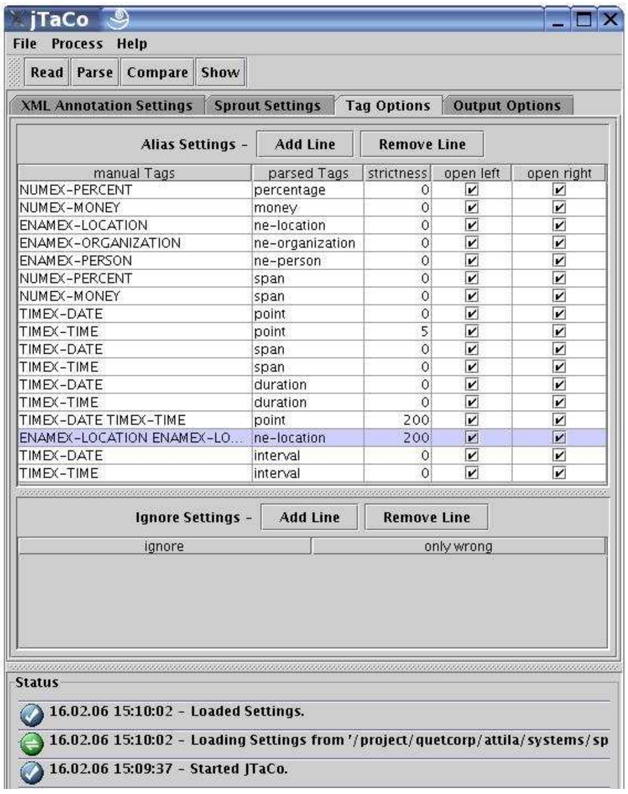
Usually, the setup uses tags which incorporate more information, and the relation used to determine entity equality between the two annotations typically depends on this information. For instance, for use with, SProUT JTaCo generates an annotation consisting of tags which are augmented with feature structure information. The equality notion of these tags is defined though unification.
An important feature of JTaCo is that the comparison can be configured to accomodate for a variety of systematic differences in annotations:

- The annotations may use different labels, differing perhaps even in granularity. E.g., one annotation might globally use the label *organisation*, while the other uses subclasses such as *university*, *government*, etc.
- The annotated entities may differ in their surface spans. E.g., one annotation might consider the expression *President Hugo Chavez* to be a named entity, while the other might exlude the title.
- One annotation may contain sequences of entities which in the other annotation correspond to one single entity. For instance, MUC will usually separate a date followed by a time into two named entities (`TIMEX-DATE` and `TIMEX-TIME`), while SProUT considers this to be one entity.

The screenshot in Figure 4 shows a part of the defined tag mappings used when comparing SProUT's annotation to the original MUC markup. Most of the mappings constitute simple entity label correspondences, e.g., a MUC `TIMEX-DATE` can correspond to a `point`, a `span`, a `duration`, or an `interval` in the annotation generated by SProUT. An entity named a `duration` by SProUT can in turn be a MUC `TIMEX-DATE` as well as a `TIMEX-TIME`. In the example settings, all of these correspondences are further 'softened' to ignore surface span discrepancies: The *open left* and *open right* switches allow for a mismatch in the `CSTART` and `CEND` features, respectively. The example settings also contain a mapping of the sequence `TIMEX-DATE` and `TIMEX-TIME` in MUC to the SProUT entity `point`. The *strictness* is a measure of how far apart these two elements are allowed to occur and still be valid elements for a sequence matched against a single `point`.

3.5. Generating a Report

Finally, JTaCo generates a report of the comparison. JTaCo can output statistical information (precision, recall, etc.) as well as detailed occurrence lists of entities that were or were not correctly identified in the parse. The settings for this processing stage determine which results are shown (e.g., for which tags) and how the information is formatted. JTaCo can export the generated reports as ASCII and as HTML tables.



manual Tags	parsed Tags	strictness	open left	open right
NUMEX-PERCENT	percentage	0	✓	✓
NUMEX-MONEY	money	0	✓	✓
ENAMEX-LOCATION	ne-location	0	✓	✓
ENAMEX-ORGANIZATION	ne-organization	0	✓	✓
ENAMEX-PERSON	ne-person	0	✓	✓
NUMEX-PERCENT	span	0	✓	✓
NUMEX-MONEY	span	0	✓	✓
TIMEX-DATE	point	0	✓	✓
TIMEX-TIME	point	5	✓	✓
TIMEX-DATE	span	0	✓	✓
TIMEX-TIME	span	0	✓	✓
TIMEX-DATE	duration	0	✓	✓
TIMEX-TIME	duration	0	✓	✓
TIMEX-DATE TIMEX-TIME	point	200	✓	✓
ENAMEX-LOCATION ENAMEX-LO...	ne-location	200	✓	✓
TIMEX-DATE	interval	0	✓	✓
TIMEX-TIME	interval	0	✓	✓

ignore	only wrong

```
16.02.06 15:10:02 - Loaded Settings.
16.02.06 15:10:02 - Loading Settings from '/project/quetcorp/attila/systems/sp
16.02.06 15:09:37 - Started JTaCo.
```

Figure 4: Definition of comparison settings in JTaCo's SProUT IDE plug-in. See Section 3.4. for a detailed explanation.

4. SProUTomat

SProUTomat, described in more detail in (Schäfer and Beck, 2006), is an automatic build, testing and evaluation tool for linguistic resources and components that has been implemented for SProUT. SProUTomat is used for daily building and testing the development and runtime system from the program and lingware source code checked out from a version control system.

4.1. Build Procedure

SProUTomat is an extension of the build mechanism for language technology components and resources we have developed for the SProUT system using Apache Ant (`http://ant.apache.org`). Ant is a standard open source tool for automatic building and packaging complex software systems. On the basis of target descriptions in an XML configuration file, Ant automatically resolves a target dependency graph and executes only the necessary targets.
Before testing and evaluating, a system has to be built, i.e., compiled from the sources checked out from the source control system. The Java program code compilation of SProUT is a straightforward task best supported by Ant. The case is, however, different for lingware sources (type hierarchy[4], tokeniser, morphology, gazetteer, XTDL grammars).
While the appropriate Java code compilation tasks know

[4]The SProUT formalism uses a subset of TDL (Krieger and Schäfer, 1994) that is compiled using the *flop* compiler of the PET system (Callmeier, 2000).

what a compiled class file is and when it has to be recompiled (source code changes, dependencies), this has to be defined explicitly for lingware resources which Ant natively is not aware of. The `uptodate` task can be used to compare source files (.tdl in the following example) against their compiled version (.grm).

```
<uptodate property="tdl_input_is_uptodate"
          srcfile="${typehierarchy}.tdl"
       targetfile="${typehierarchy}.grm"/>
```

For each of the different lingware types, these source file dependencies are defined as are the calls to the dedicated SProUT compilers and parameters for their compilation.
Lingware-specific targets have common parameters and properties like `"lang"`, `"project"` or the lingware type that are used to locate, e.g., the source and compiled files in the hierarchically defined directory trees or `"charset"` to specify encodings for source files to read.

```
<!--usage : ant compile_ne -Dlang=en -->
<target name="compile_ne" depends="jar"
 description="Compile NER grammar.">
  <property name="lang" value="en"/>
  <property name="project" value=""/>
  <property name="charset" value="utf-8"/>
  <!-- compile type hierarchy -->
  <antcall target="compile_tdl"/>
  <!-- compile tokeniser -->
  <antcall target="compile_tokenclass"/>
  <!-- compile gazetteer -->
  <antcall target="compile_gazetteer"/>
  <!-- compile XTDL grammar for NER -->
  <antcall target="compile_grammar"/>
</target>
```

Figure 5: A sample target definition: named entity grammar compilation.

Dependencies between different lingware types are handled by calls to defined sub-targets. Figure 5 shows the definition of the `compile_ne` target that calls four other compilation sub-targets. Each subtarget compiles only when necessary, and the `compile_ne` target itself depends on the `jar` target that provides working and up-to-date SProUT lingware compilers.
Besides the program and lingware compilation, many other targets exist, e.g., to generate documentation, package runtime systems, start the integrated development environment, etc.
Thus, using a single command, it is possible to compile the whole system including code and all dependent available linguistic resources, or to update it after changes in the sources.

4.2. Test and Evaluation

When SProUTomat is started, it first updates all program and lingware sources from the version control system, and compiles them. For each language resource to test, a reference text is then analysed by the SProUT runtime system. This checks for consistent (re)sources. The next step is comparison of the generated named entity and information extraction annotation against a gold standard. SProUTomat uses the batch version of JTaCo for the automatic evaluation and computation of precision, recall and f-measure. For English, the annotated corpus is taken from the MUC evaluation data. For other languages for which no MUC annotations exist (e.g., German), a manually developed corpus is employed.

4.2.1. Report

Finally, a report is generated and emailed to the developers with an overall status (OK or ERROR) for quick information. The report also contains diagrams consisting of precision, recall and f-measure curves since beginning of regular measurements per language that visually give a quick overview of the resource development progress over time (cf. Figure 6). To this end, the evaluation numbers are also added to a global evaluation database.

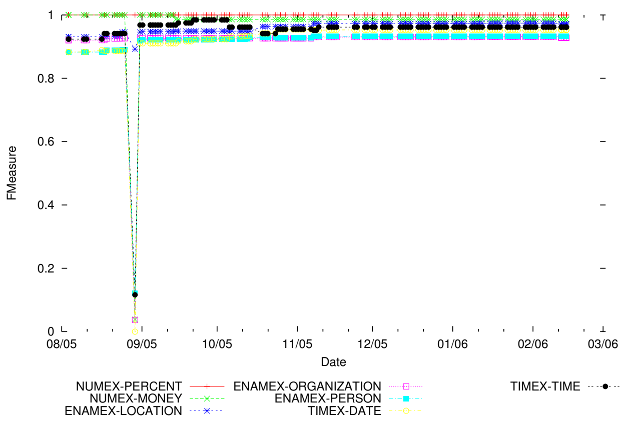


Figure 6: F-measure curves of the English MUC-compatible NER grammar collected by SProUTomat from 08/05 to 03/06. The drop in August/September was caused by a code change not followed by an immediate adaptation of the lingware.

5. Summary

We have presented a comprehensive framework for automatically testing and evaluating multilingual linguistic resources and language technology components. The system is in daily use since March 2005 and successfully helps to maintain the quality and reliability of the multilingual language processor with its various resources that are developed by many authors and used in several projects. The framework greatly helps to improve and accelerate the development - evaluation/comparison - refinement cycle, gives motivating feedback (such as raising recall and precision curves over time) and thus provides continuous quality assurance for a complex natural language processing system.

6. Acknowledgements

We would like to thank Daniel Beck for helping to implement SProUTomat, the SProUT grammar developers for their feedback, Witold Drożdżyński for extending the SProUT API to our needs and the reviewers for helpful comments.

This work has been supported by research grants from the German Federal Ministry of Education and Research in the context of the projects QUETAL (FKZ 01 IW C02) and COLLATE (FKZ 01 IN A01).

7. References

Christian Bering, Witold Drozdzyski, Gregor Erbach, Clara Guasch, Petr Homola, Sabine Lehmann, Hong Li, Hans-Ulrich Krieger, Jakub Piskorski, Ulrich Schäfer, Atsuko Shimada, Melanie Siegel, Feiyu Xu, and Dorothee Ziegler-Eisele. 2003. Corpora and evaluation tools for multilingual named entity grammar development. In *Proceedings of Multilingual Corpora Workshop at Corpus Linguistics*, pages 42–52, Lancaster, UK.

Christian Bering, 2004. *JTaCo User Guide*. Saarbrücken, Germany. Saarland University, Computational Linguistics Department.

Ulrich Callmeier. 2000. PET – A platform for experimentation with efficient HPSG processing techniques. *Natural Language Engineering*, 6(1):99–108.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. 2002. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*, Philadelphia, PA.

Witold Drozdzynski, Hans-Ulrich Krieger, Jakub Piskorski, Ulrich Schäfer, and Feiyu Xu. 2004. Shallow processing with unification and typed feature structures – foundations and applications. *Künstliche Intelligenz*, 2004(1):17–23. Available online.

Ralph Grishman and Beth Sundheim. 1996. Message understanding conference - 6: A brief history. In *Proceedings of COLING-96*, pages 466–471, Copenhagen, Denmark.

Hans-Ulrich Krieger and Ulrich Schäfer. 1994. TDL – a type description language for constraint-based grammars. In *Proceedings of COLING-94*, pages 893–899.

Kiyong Lee, Lou Burnard, Laurent Romary, Eric de la Clergerie, Ulrich Schäfer, Thierry Declerck, Syd Bauman, Harry Bunt, Lionel Clément, Tomaz Erjavec, Azim Roussanaly, and Claude Roux. 2004. Towards an international standard on feature structure representation (2). In *Proceedings of the LREC-2004 workshop on A Registry of Linguistic Data Categories within an Integrated Language Resources Repository Area*, pages 63–70, Lisbon, Portugal.

Ulrich Schäfer and Daniel Beck. 2006. Automatic testing and evaluation of multilingual language technology resources and components. In *Proceedings of the 5th International Conference on Language Resources and Evaluation LREC-2006*, Genoa, Italy.