Cerstin Mahlow
Michael Piotrowski (Eds.)

Communications in Computer and Information Science 41

State of the Art in Computational Morphology

Workshop on Systems and Frameworks
for Computational Morphology, SFCM 2009
Zurich, Switzerland, September 2009, Proceedings

Springer

Communications
in Computer and Information Science 41

Cerstin Mahlow Michael Piotrowski (Eds.)

State of the Art in Computational Morphology

Workshop on Systems and Frameworks
for Computational Morphology, SFCM 2009
Zurich, Switzerland, September 4, 2009
Proceedings

Springer

Volume Editors

Cerstin Mahlow
University of Zurich
Zurich, Switzerland
E-mail: mahlow@cl.uzh.ch

Michael Piotrowski
University of Zurich
Zurich, Switzerland
E-mail: mxp@cl.uzh.ch

Library of Congress Control Number: Applied for

CR Subject Classification (1998): J.5, H.3.1, F.4.2, F.4.3, I.2.7, H.5.2

ISSN 1865-0929
ISBN-10 3-642-04130-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-04130-3 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12745241 06/3180 5 4 3 2 1 0

Preface

From the point of view of computational linguistics, morphological resources are the basis for all higher-level applications. This is especially true for languages with a rich morphology, such as German or Finnish. A morphology component should thus be capable of analyzing single word forms as well as whole corpora. For many practical applications, not only morphological analysis, but also generation is required, i.e., the production of surfaces corresponding to specific categories.

Apart from uses in computational linguistics, there are also numerous practical applications that either require morphological analysis and generation or that can greatly benefit from it, for example, in text processing, user interfaces, or information retrieval. These applications have specific requirements for morphological components, including requirements from software engineering, such as programming interfaces or robustness.

In 1994, the First Morpholympics took place at the University of Erlangen-Nuremberg, a competition between several systems for the analysis and generation of German word forms. Eight systems participated in the First Morpholympics; the conference proceedings [1] thus give a very good overview of the state of the art in computational morphology for German as of 1994.

Today, 15 years later, some of the systems that participated in the Morpholympics still exist and are being maintained. However, there are also new developments in the field of computational morphology, for German and for other languages. Unfortunately, publications on morphologic analysis and generation are spread over many different conferences and journals, so that it is difficult to get an overview of the current state of the art and of the available systems. One of our goals for the Workshop on Systems and Frameworks for Computational Morphology (SFCM 2009) was therefore to bring together researchers, developers, and maintainers of morphology systems for German and of frameworks for computational morphology from academia and industry, in order to produce an up-to-date overview of available systems for German.

As in many areas of computational linguistics, there are rule-based and statistical approaches to morphological analysis; SFCM 2009 focused on systems and frameworks based on linguistic principles and providing linguistically motivated analyses and/or generation on the basis of linguistic categories.

One factor for this decision was our own experience in implementing rule-based morphological analyzers, so we know that they are able to deliver detailed, structured analyses (see [2]). The possibility to draw upon the morphological processes of inflection, derivation, and compounding involved when analyzing or generating word forms, the respective parse and generation trees, and certain elements of the category, is, in our view, a potential advantage when compared with statistically created results.

The second and deciding factor is that, based on the results of Morpho Challenge[1], we have come to the conclusion that statistical morphological analyzers are not yet able to deliver the quality of results required for practical applications. In the Morpho Challenge morpheme analysis task, the analyses proposed by the participants' algorithms are compared against a linguistic gold standard. At Morpho Challenge 2008 [3], the best system for German achieved an F-measure of 54.06%. The best recall value was 59.51% (this system achieved 49.53% precision), the best result for precision was 87.92% (with 7.44% recall).[2] These figures are much too low to consider the systems as suitable for use in most types of applications, and in interactive applications in particular. If we compare the results of Morpho Challenge with the figures reported from the First Morpholympics [4], the decision to focus on rule-based systems suggests itself.

In the call for papers for this workshop we asked for contributions on actual, working systems and frameworks of at least prototype quality. To ensure fruitful discussions among workshop participants, we asked that submissions on concrete morphology systems should be for German; submissions on morphological frameworks were considered relevant if the framework can be used to implement components for different languages.

The workshop thus had three main goals:

- To stimulate discussion among researchers and developers and to offer an up-to-date overview of available systems for German morphology which provide deep analyses and are suitable for generating specific word forms.
- To stimulate discussion among developers of general frameworks that can be used to implement morphological components for several languages.
- To discuss aspects of evaluation of morphology systems and possible future competitions or tasks, such as a new edition of the Morpholympics.

Based on the number of submissions and the number of participants at the workshop we can definitely state that the topic of the workshop has met with great interest from the community, both from academia and industry. We received 16 submissions, of which 9 were accepted after a thorough review by the members of the Program Committee and additional reviewers. The peer review process was double-blind, and each paper received at least three reviews.

The discussions after the talks and during the demo sessions, as well as the final plenum, showed the interest in and the need and the requirements for further efforts in the field of computational morphology. We will maintain the website for this workshop at `http://sfcm2009.org`. Here you can find additional material not included in the proceedings. If there is a follow-up to this workshop—whether in a similar format or in the form of a competition—it will also be announced on this site.

[1] Morpho Challenge is a shared task and conference for the evaluation of statistical morphological components based on unsupervised machine-learning.

[2] See `http://www.cis.hut.fi/morphochallenge2008/` for details. The results of Morpho Challenge 2009 were not yet available at the time of this writing.

This book starts with a theory-oriented paper by Thomas Hanneforth ("Using Ranked Semirings for Representing Morphology Automata"), proposing complex weight structures to represent morphological analyzers.

The following three papers report on frameworks: Johannes Handl, Besim Kabashi, Thomas Proisl, and Carsten Weber ("JSLIM – Computational Morphology in the Framework of the SLIM Theory of Language") present a recent implementation of the SLIM theory, based on Left-Associative Grammar [5]. This framework allows the implementation of morphological analyzers and generators for different languages. Krister Lindén, Miikka Silfverberg, and Tommi Pirinen ("HFST Tools for Morphology – An Efficient Open Source Package for Construction of Morphological Analyzers") present a recent implementation of Two-Level Morphology [6]. Both frameworks (JSLIM and HFST) are intended to be distributed as open-source software, and both papers report on actually implemented systems for several languages with a coverage permitting the analysis of real-world texts. This is a very positive trend, which—we hope—will encourage the development of concrete analyzers and generators using the knowledge of the community and coordinating the efforts. Thomas Hanneforth ("fsm2 – A Scripting Language for Creating Weighted Finite-State Morphologies") reports on a scripting language for creating and manipulating morphological analyzers for different languages based on weighted semirings.

The following three papers report on morphological systems for German. Andrea Zielinski, Christian Simon, and Tilman Wittl ("Morphisto – Service-Oriented Open Source Morphology for German") present their recent efforts in developing an open-source analyzer and generator for German. It was initially developed within the TextGrid project—a modular platform for collaborative textual editing for researchers in philology, linguistics, and related fields. Morphisto is based on SMOR and the SFST tools [7], offering a comprehensive free lexicon. Heinz Dieter Maas, Christoph Rösener, and Axel Theofilidis ("Morphosyntactic and Semantic Analysis of Text: The MPRO Tagging Procedure") present further details of a system which already participated in the first Morpholympics in 1994. MPRO is still maintained and being used in several applications. Similarly, the system presented by Pius ten Hacken ("Word Manager") has been maintained for more than 15 years and is being used in various environments.

The last two papers deal with unknown words. The first one, by Stephan Bopp and Sandro Pedrazzini ("Morphological Analysis Using Linguistically Motivated Decomposition of Unknown Words") is a practical application of Word Manager described in the preceding paper. While Bopp and Pedrazzini concentrate on the decomposition of complex German compounds using an existing morphological system, the paper of Krister Lindén and Jussi Tuovila ("Corpus-based Lexeme Ranking for Morphological Guessers") is concerned with adding new words to the lexicon of a morphological component. Their approach is intended to help the lexicographer by providing automatically generated suggestions for lexemes.

In summary, these contributions show that high-quality research is being conducted in the area of rule-based computational morphology, and that there are further developments of mature systems, new implementations based on established theoretical frameworks, and new approaches to the problems of morphology. We also see a trend towards open-source developments, which we find very promising. Open-source projects allow

the collaboration of researchers interested in morphological systems which fulfill high demands on performance and quality. It should not be necessary to develop a morphological analyzer from scratch if one is needed for a project. Making high-quality morphological resources freely available will help to advance the state of the art and allow the development of high-quality real-world applications. Useful applications will demonstrate to a broad audience that computational morphology (and natural language processing in general) is not an esoteric boondoggle, but an actual science with tangible benefits for society, which are good reason for publicly funding research in this area.

We would like to thank the authors for their contributions to the workshop and to this book. We also thank the reviewers for their effort and for their constructive feedback, encouraging and helping the authors to improve their papers. The submission and reviewing process and the compilation of the proceedings was supported by the EasyChair system. We thank Stefan Göller, the editor of the series ("Communications in Computer and Information Science") (CCIS), and Springer for publishing the proceedings of SFCM 2009. We are grateful for the financial support given by the Institute of Computational Linguistics at the University of Zurich and by the German Society for Computational Linguistics and Language Technology (GSCL). Last, but not least, we thank Roland Hausser for encouraging us to organize the SFCM 2009 workshop as an almost successor of the First Morpholympics in 1994, and Norbert Fuchs for providing many helpful hints during the entire organization process.

July 2009

Cerstin Mahlow
Michael Piotrowski

References

1. Hausser, R.: Linguistische Verifikation. Dokumentation zur Ersten Morpholympics. Niemeyer, Tübingen (1996)
2. Mahlow, C., Piotrowski, M.: SMM: Detailed, structured morphological analysis for Spanish. Polibits. Computer science and computer engineering with applications (39) (2009)
3. Kurimo, M., Varjokallio, M.: Unsupervised morpheme analysis evaluation by a comparison to a linguistic gold standard – Morpho Challenge 2008. In: Workshop of the Cross-Language Evaluation Forum (CLEF 2008). (2008)
4. Lenders, W., Bátori, I., Dogil, G., Görz, G., Seewald, U.: Stellungnahme der Jury für die Morpholympics 94. In Hausser, R., ed.: Linguistische Verifikation. Dokumentation zur Ersten Morpholympics 1994. Niemeyer, Tübingen (1996) 15–24
5. Hausser, R.: Foundations of Computational Linguistics: Human-Computer Communication in Natural Language. 2nd rev. and ext. edn. Springer, Heidelberg (2001)
6. Koskenniemi, K.: Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production. PhD thesis, University of Helsinki (1983)
7. Schmid, H., Fitschen, A., Heid, U.: A German computational morphology covering derivation, composition, and inflection. In: IVth International Conference on Language Resources and Evaluation (LREC 2004). (2004) 1263–1266

Organization

The Workshop on Systems and Frameworks for Computational Morphology (SFCM 2009) was organized by Cerstin Mahlow and Michael Piotrowski. The workshop was held at the University of Zurich.

Program Chairs

Cerstin Mahlow	University of Zurich, Switzerland
Michael Piotrowski	University of Zurich, Switzerland

Program Committee

Simon Clematide	University of Zurich, Switzerland
Thomas Hanneforth	University of Potsdam, Germany
Roland Hausser	Friedrich-Alexander University of Erlangen-Nuremberg, Germany
Lauri Karttunen	PARC, Palo Alto, USA
Kimmo Koskenniemi	University of Helsinki, Finland
Winfried Lenders	University of Bonn, Germany
Krister Lindén	University of Helsinki, Finland
Anke Lüdeling	Humboldt University Berlin, Germany
Cerstin Mahlow	University of Zurich, Switzerland
Günter Neumann	DFKI Saarbrücken, Germany
Michael Piotrowski	University of Zurich, Switzerland
Helmut Schmid	University of Stuttgart, Germany
Angelika Storrer	University of Dortmund, Germany
Martin Volk	University of Zurich, Switzerland
Shuly Wintner	University of Haifa, Israel
Andrea Zielinski	FIZ Karlsruhe, Germany

Additional Reviewers

Bruno Cartoni	University of Geneva, Switzerland
Johannes Handl	Friedrich-Alexander University of Erlangen-Nuremberg, Germany
Besim Kabashi	Friedrich-Alexander University of Erlangen-Nuremberg, Germany
Thomas Proisl	Friedrich-Alexander University of Erlangen-Nuremberg, Germany

Luzius Thöny	University of Zurich, Switzerland
Carsten Weber	Friedrich-Alexander University of Erlangen-Nuremberg, Germany

Local Organization

Cerstin Mahlow	University of Zurich, Switzerland
Michael Piotrowski	University of Zurich, Switzerland
Nancy Renning	University of Zurich, Switzerland

Sponsoring Institutions

Institute of Computational Linguistics, University of Zurich
German Society for Computational Linguistics and Language Technology (GSCL)

Table of Contents

Using Ranked Semirings for Representing Morphology Automata

Thomas Hanneforth

Department for Linguistics, University of Potsdam, Germany

Abstract. We propose a class of complex weight structures called *ranked semirings* for the compact representation of morphological analysers based on weighted finite-state automata. In an experiment, we compare this compact representation with the conventional representation based on *letter transducers*.

1 Introduction

Using finite-state transducers has a long-standing tradition in computational morphology, since their formal properties allow for creating complex systems in a modular fashion. Most systems are based on a variant of finite-state transducer, called *letter transducer*. A letter transducer (henceforth *LFST*) is a finite-state transducer $T = \langle Q, \Sigma, \Gamma, q_0, F, E \rangle$ with input alphabet Σ and output alphabet Γ, where the set of transitions E is defined as a subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times Q$. That means that only single symbols (in opposition to strings) or ε may label a transition between two states. This restriction – which doesn't limit the generative capacity compared to unrestricted finite-state transducers – facilitates the efficient implementation of these automata, since alphabet symbols are most often represented as integers which can be very efficiently processed. Furthermore, the avoidance of strings or symbol vectors leads to simpler algorithms for composition and intersection of automata.

On the other hand, letter transducers tend to be less compact and efficient. In computational morphology systems (CMS), the analyses computed for a given input word are in general much longer than the input word itself. Typically, the CMS will insert morpheme boundaries and enrich the input with morpho-syntactic information like word class, features, etc. In a letter transducer, where a transition can only emit a single symbol, the length difference between input and output is usually compensated by inserting input-ε-transitions.[1] Even in certain circumstances where it is guaranteed to move these input-ε-transitions towards the end of an accepting path[2], they nevertheless increase the amount of non-deterministic computation. Figure 1 shows a LFST for the German definite and indefinite articles.[3] Furthermore, optimisation transformations like determinisation and minimisation become harder and less advantageous:

[1] In *Two-Level*-systems ([1]) which are based on parallel intersection of letter transducers which represent different constraints, ε is disallowed and a regular symbol (e.g., "0") with a special meaning is used instead.

[2] For example, for acyclic transducers representing lexicons.

[3] This LFST is already optimised by using the *encoded minimization* technique: symbol pairs are treated as symbols taken from a new alphabet; after that, the LFST is determinised and minimised as an ordinary finite-state acceptor.

C. Mahlow and M. Piotrowski (Eds.): SFCM 2009, CCIS 41, pp. 1–9, 2009.

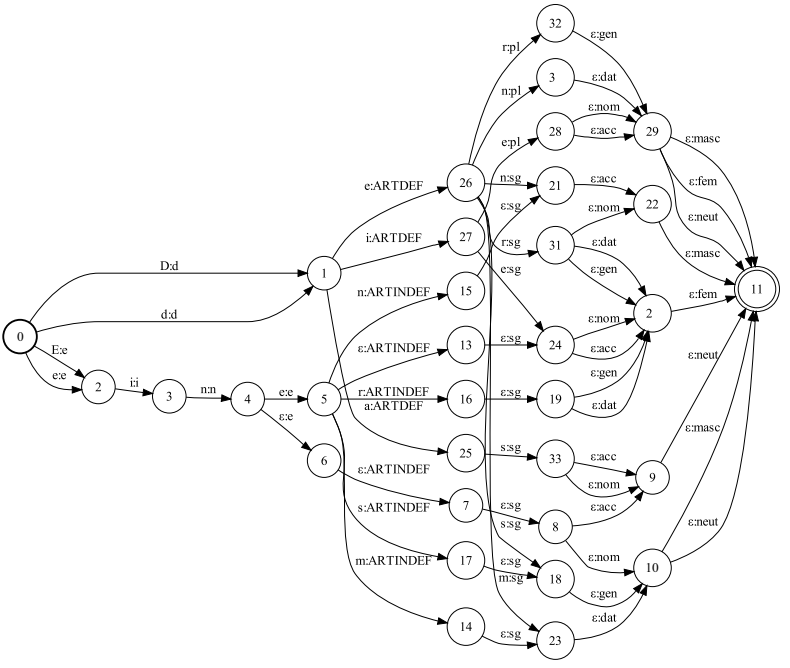

Fig. 1. Letter transducer for German articles

- Determinisation of string transducers (see [2]) (of which letter transducers are a special case) is based on an extended powerset construction which keeps track – for every component state in the state set – of the symbols already outputted and the symbols to be outputted later (the *delayed symbols*). While computing a transition from state set P to state set Q for an input symbol a in the equivalent deterministic FST, the determinisation algorithm can safely output the *longest common prefix* of all the delayed symbol strings w for a pair $\langle p, w\rangle$ in state set P, each one concatenated with the output symbols of the a-transitions leaving state p in the non-deterministic FST. (see [2] for details). In many practical cases encountered in morphology processing, this prefix $v = v_1 \ldots v_k$ is no longer a single symbol. When being restricted to letter transducers, the only possible solution is to output the first symbol v_1 of v together with the input symbol a and then output the rest of v by using input-ε-transitions. Again, this move make the resulting LFST technically non-deterministic and increases its size unnecessarily.
- Minimisation reduces the space requirements of a finite-state transducer by collapsing equivalent path suffixes. Minimisation is mainly based on a equivalence transformation called *pushing* ([3]) in which output strings are pushed as far as possible towards the initial state of the transducer. Technically, for each state p of the transducer, an *(output) string potential* $Pot[p]$ is computed with the help of a distance computation algorithm. After that, the output w of each transition $p \xrightarrow{a} q$ of the original machine is *reweighted* based on the string potentials of p and q. Again, we are faced with a similar problem as in the determinisation case above: the need

may arise to output more than a single symbol when following the transition, even if the original FST has been a letter transducer.

The article is structured as follows: Section 2 will provide the necessary technical definitions, while Section 3 introduces a complex weight structure for weighted finite-state automata called *ranked semiring*. The article concludes with an experiment and a discussion on implementational issues.

2 Definitions

Readers familiar with the subject of semiring-weighted finite-state automata may skip this section.

Before we start to define weighted automata, we need a suitable weight structure for them.

Definition 1 (Semiring). *An algebraic structure $\mathscr{K} = \langle W, \oplus, \otimes, \overline{0}, \overline{1} \rangle$ is a semiring [4] if it fulfills the following conditions:*

1. $\langle W, \oplus, \overline{0} \rangle$ *is a commutative monoid with* $\overline{0}$ *as the identity element for* $\oplus$.
2. $\langle W, \otimes, \overline{1} \rangle$ *is a monoid with* $\overline{1}$ *as the identity element for* $\otimes$.
3. $\otimes$ *distributes over* $\oplus$:
   $\forall x, y, z \in W : x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$ *(left distributivity)*
   $\forall x, y, z \in W : (y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x)$ *(right distributivity)*
4. $\overline{0}$ *is an annihilator for* $\otimes$: $\forall w \in W, w \otimes \overline{0} = \overline{0} \otimes w = \overline{0}$.

Commonly used semirings are for example the *tropical semiring*

$\mathscr{T} = \langle R_0^+ \cup \{+\infty\}, min, +, +\infty, 0 \rangle$ for positive real numbers and the *Viterbi semiring* $\mathscr{V} = \langle [0 \ldots 1], max, \cdot, 0, 1 \rangle$ for probabilities.

In the following, a semiring $\mathscr{K}$ is identified with its carrier set W.

Definition 2 (Semiring properties). *Let $\mathscr{K} = \langle W, \oplus, \otimes, \overline{0}, \overline{1} \rangle$ be a semiring.*

- $\mathscr{K}$ *is called* ***idempotent*** *if* $\forall a \in \mathscr{K} : a \oplus a = a$.
- $\mathscr{K}$ *has the* ***path property*** *if* $\forall a, b \in \mathscr{K} : a \oplus b = a$ *or* $a \oplus b = b$.

Weighted finite-state automata are an extension to classical finite-state automata ([5]) in which transitions and final states are augmented with weights taken from a semiring.

Definition 3 (Weighted finite-state acceptor). *A* ***weighted finite-state acceptor*** *(henceforth WFSA, cf. [6])*
$\mathfrak{A} = \langle \Sigma, Q, q_0, F, E, \lambda, \rho \rangle$ *over a semiring $\mathscr{K}$ is a 7-tuple with*

1. Σ, *the finite input alphabet,*
2. Q, *the finite set of states,*
3. $q_0 \in Q$, *the start state,*
4. $F \subseteq Q$, *the set of final states,*
5. $E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \mathscr{K} \times Q$, *the set of transitions,*
6. $\lambda \in \mathscr{K}$, *the initial weight and*
7. $\rho : F \to \mathscr{K}$, *the final weight function mapping final states to elements in $\mathscr{K}$.*

The following definition functionally relates strings accepted (or generated) by a WFSA with semirings weights:

Definition 4 (Weight associated with a string). *Given a WFSA A, let a* path $\pi = t_1 t_2 \ldots t_k$ *be a sequence of adjacent transitions. Let $\Pi(q,x,p)$ for $x \in \Sigma^*$ be the set of paths from state $q \in Q$ to state $p \in Q$ such that the concatenation of input symbols in each $\pi \in \Pi(q,x,p)$ equals x. Let $\omega(\pi)$ the $\otimes$-multiplication of all transitions weights in a path π: $\omega(\pi) = w[t_1] \otimes w[t_2] \otimes \ldots \otimes w[t_k]$. The* ***weight associated with an input string*** *$x \in \Sigma^*$* ***wrt a WSFA*** *A – denoted by $[\![x]\!]_A$ – is computed by the following equation:*

$$[\![x]\!]_A = \bigoplus_{\substack{q \in F, \\ \pi \in \Pi(q_0,x,q)}} \lambda \otimes \omega(\pi) \otimes \rho(q)$$

In definition 4, weights along a path π are combined with the $\otimes$-operation, that is, they are *multiplied* in the Viterbi semiring or *added* in the tropical semiring. Different paths π for the input x are then combined with the $\oplus$-operation, which for example computes the *maximum* in the Viterbi semiring or the *minimum* in the tropical semiring. The requirement that $\oplus$ is commutative (see definition 1) ensures that the order in which the paths π are processed is irrelevant. Note that the $\oplus$-operation in definition 4 plays only a role if there are several accepting paths for input string x, that is, if the WFSA A is non-deterministic.

We have now everything at hand to define more complex semiring structures.

3 Ranked Semirings

Morphology outputs are most often strings, that is, symbol sequences, over an output alphabet Δ. To treat them as weights, we need to define a suitable semiring structure for them. One possibility is a *string semiring* $\mathscr{S} = \langle \Delta^* \cup \{s_\infty\}, \wedge, \cdot, s_\infty, \varepsilon \rangle$ [2]. The abstract addition operation $\oplus$ is mapped to the computation of the *longest common prefix* – denoted by $\wedge$ – of two strings; for example: $ARTDEF\#sg \wedge ARTDEF\#pl = ARTDEF\#$. The identity element of $\wedge$ is s_∞, that is, $\forall x \in \Delta^* : x \wedge s_\infty = x$ (that means every string $x \in \Delta^*$ is a prefix of s_∞). Note that $\wedge$ is idempotent, that is, for every string x, $x \wedge x = x$. As already discussed in the previous section, the longest common prefix operation $\wedge$ is very useful for computing maximal outputs during determinisation and for realising outputs as early as possible (pushing).

The place of the abstract multiplication operation $\otimes$ is filled in $\mathscr{S}$ by string concatenation $\cdot$ with the empty string ε as the identity element.

Parametrising definition 4 with $\mathscr{S}$, we arrive at a WFSA in which string weights are concatenated along a path in the WSFA, as desired:

$$[\![x]\!]_A = \bigwedge_{\substack{q \in F, \\ \pi \in \Pi(q_0,x,q)}} \lambda \cdot \omega(\pi) \cdot \rho(q) \tag{1}$$

Characteristic for morphological analysis systems is the fact that in many cases input words are mapped to several analyses. For example, the German definite article *die*

has 8 different analyses which differ in gender, number and case. Allowing that would result in a nondeterministic WFSA with degraded efficiency. Furthermore, in the case of several accepting paths for a word w, equation 1 would compute $\wedge$ over all strings weights along the different paths for w. In the example case of the article *die*, only the longest common prefix *d ARTDEF* of all of them would be the result[4].

To solve this problem, it was proposed in [2] to allow a maximum of p different outputs at a final state q of a WFSA. This leads to the definition of *p-subsequential WFSAs*, that is, deterministic WFSAs where the output function ρ is altered in the following way:

$$\rho : F \mapsto (\Delta^*)^k, 1 \leq k \leq p. \tag{2}$$

Equation 1 needs to be changed to reflect the new definition of ρ correctly:

$$[\![x]\!]_A = \bigcup_{\substack{q \in F, v \in \rho(q) \\ \pi \in \Pi(q_0, x, q)}} \{\lambda \cdot \omega(\pi) \cdot v\} \tag{3}$$

A second solution is to keep definition 4 and "raise" the definition of the semiring in an appropriate way. We will pursue this possibility later.

In many applications, analyses computed by CMSs need to be ranked somehow. In [7] and [8], it was proposed to measure the morphological complexity of a word by counting the occurrences of different morpheme boundary symbols inside an analysis. The analysis with the lowest score, that is, with the smallest number of morpheme boundaries, is then selected. Technically, this is achieved by creating a weighted letter transducer over the tropical semiring and assigning penalty weights to the boundary symbols, either dependent on the morphological construction or globally. After having looked up a word in the transducer, a *best path* operation ([6]) is performed. Since the abstract multiplication operation of the tropical semiring corresponds to addition, penalties along a path are added.

Another possibility is enriching the morphology transducer with probabilities computed from a corpus by using the Viterbi semiring.

To compare weights with each other, is is necessary to define some order amongst them. Idempotent semirings have an associated order called the *natural order* (see [6]). If $\mathscr{K}$ is an idempotent semiring, for any two elements x and $y \in \mathscr{K}$, we define: $x \leq_{\mathscr{K}} y$ if $x \oplus y = x$. It can be easily verified that $\leq_{\mathscr{K}}$ defines a partial, that is, reflexive, transitive and antisymmetric order.

Example 1 (Natural order in idempotent semirings)

- In the tropical semiring $\mathscr{T}$, $2 \leq_{\mathscr{T}} 5$, since $2 \; min \; 5 = 2$
- In the Viterbi semiring $\mathscr{V}$, $0.75 \leq_{\mathscr{V}} 0.5$, since $0.75 \; max \; 0.5 = 0.75$
- In the string semiring $\mathscr{S}$, $cat \leq_{\mathscr{S}} cats$, since $cat \wedge cats = cat$

If an idempotent semiring $\mathscr{K}$ has in addition the path property, a WFSA over $\mathscr{K}$ can be used in best path operations. Remember that the path property says that for any two weights x and $y \in \mathscr{K}$, $x \oplus y = x$ or $x \oplus y = y$. That is, the $\oplus$-comparison of two weights

[4] We treat *d* as the lemma of all German definite article forms *der*, *die*, *das*, etc.

x and y computed along two paths in a WSFA A either yields the path corresponding to x or the path corresponding to y.

Observe that the string semiring does not have the path property, as already discussed above.

It is possible to define complex semirings $\mathscr{K}_i$ by combining simpler semirings. If the resulting semiring should be useful for best path operations, a ranking among the individual sub-semirings must be settled. The following definition of a *ranked tuple semiring* accomplishes that:

Definition 5 (Ranked tuple semiring). *Let $\mathscr{K}_i$ be idempotent semirings ($1 \leq i \leq k$) for some constant $k \geq 1$.*

Let $\overrightarrow{x} = \langle x_1, x_2, \ldots x_k \rangle$ and $\overrightarrow{y} = \langle y_1, y_2, \ldots y_k \rangle$ be two semiring tuples, with $x_i, y_i \in \mathscr{K}_i$. Define a new structure $\overrightarrow{\mathscr{K}} = \langle \mathscr{K}_1 \times \mathscr{K}_2 \times \ldots \times \mathscr{K}_k, \overrightarrow{\oplus}_k, \overrightarrow{\otimes}_k, \overline{0}_k, \overline{1}_k \rangle$ where:

1. $\overline{0}_k = \langle \overline{0}_{\mathscr{K}_1}, \ldots, \overline{0}_{\mathscr{K}_k} \rangle$
2. $\overline{1}_k = \langle \overline{1}_{\mathscr{K}_1}, \ldots, \overline{1}_{\mathscr{K}_k} \rangle$
3. $\overrightarrow{x} \overrightarrow{\otimes}_k \overrightarrow{y} = \langle x_1 \otimes_{\mathscr{K}_1} y_1, \ldots, x_k \otimes_{\mathscr{K}_k} y_k \rangle$
4. $$\overrightarrow{x} \overrightarrow{\oplus}_k \overrightarrow{y} = \begin{cases} \overrightarrow{x} & \exists j \leq k, \forall i, 1 \leq i < j : x_i = y_i \wedge x_j \oplus_{\mathscr{K}_j} y_j = x_j \\ \overrightarrow{y} & \text{otherwise} \end{cases}$$

Items 1. to 3. in definition 5 are straightforward. The definition of the $\oplus$-operation constitutes some sort of *lexicographic natural order* over the k-tuples. As can immediately be seen from the definition of $\overrightarrow{\oplus}_k$, the ranked tuple semiring is idempotent by letting $\overrightarrow{y} = \overrightarrow{x}$. Moreover, if the sub-semirings have the path property, $\overrightarrow{\mathscr{K}}$ will have it too.

Example 2 demonstrates the effect of $\overrightarrow{\oplus}_k$.

Example 2 ($\overrightarrow{\oplus}_k$). Consider a complex semiring $\mathscr{T} \times \mathscr{T} \times \mathscr{T}$:

- $\langle 2,3,4 \rangle \overrightarrow{\oplus}_3 \langle 2,3,2 \rangle = \langle 2,3,2 \rangle$
- $\langle 1,2,3 \rangle \overrightarrow{\oplus}_3 \langle 3,2,1 \rangle = \langle 1,2,3 \rangle$
- $\langle 1,2,3 \rangle \overrightarrow{\oplus}_3 \langle 1,2,3 \rangle = \langle 1,2,3 \rangle$

Ranked tuple semirings allow arbitrary combinations of idempotent sub-semirings, for example, for counting and weighting morphemes (with tropical semirings) or maximising probabilities (with Viterbi semirings).

Since the class of string semirings does not have the path property, we may handle a string sub-semiring within a ranked tuple semiring in two possible ways:

1. Reintroduce subsequentiality and change eq. (3) accordingly.
2. Keep definition 4 defining the weight of an input string and define a new class of *ranked string semirings* with tuple sets as weights.

We will explicate the last point in greater detail.

As already discussed, since morphological analysis systems need to map input words to *sets* of (complex) analyses, the weights in a ranked string semiring must be *sets* of pairs $\langle \mathscr{S}, \mathscr{K} \rangle$. Accordingly, the carrier set of the ranked string semiring must consist out of all subsets of $\mathscr{S} \times \mathscr{K}$, that is, its powerset.

Definition 6 (Ranked string (powerset) semiring). *Let* $\overrightarrow{\mathscr{K}} = \langle W, \oplus_{\overrightarrow{\mathscr{K}}}, \otimes_{\overrightarrow{\mathscr{K}}}, \overline{0}, \overline{1} \rangle$ *be a ranked tuple semiring.*

A ***ranked string semiring*** $\overrightarrow{\mathscr{S}}$ *is a structure* $\langle 2^{\mathscr{S} \times \overrightarrow{\mathscr{K}}}, \oplus_{\mathscr{S} \times \overrightarrow{\mathscr{K}}}, \otimes_{\mathscr{S} \times \overrightarrow{\mathscr{K}}}, \emptyset, \{\langle \varepsilon, \overline{1} \rangle\} \rangle$ *with* $(X, Y \in 2^{\mathscr{S} \times \overrightarrow{\mathscr{K}}})$*:*

$$
\begin{aligned}
X \otimes_{\mathscr{S} \times \overrightarrow{\mathscr{K}}} Y &= \{ \langle uw, v \otimes_{\overrightarrow{\mathscr{K}}} z \rangle \mid \langle u, v \rangle \in X \wedge \langle w, z \rangle \in Y \} \\
X \oplus_{\mathscr{S} \times \overrightarrow{\mathscr{K}}} Y &= \{ \langle u, v \oplus_{\overrightarrow{\mathscr{K}}} w \rangle \mid \exists u : \langle u, v \rangle \in X \wedge \langle u, w \rangle \in Y \} \\
&\quad \cup \{ \langle u, v \rangle \in X \mid \neg \exists w : \langle u, w \rangle \in Y \} \\
&\quad \cup \{ \langle u, w \rangle \in Y \mid \neg \exists v : \langle u, v \rangle \in X \}
\end{aligned}
$$

Example 3 $(\oplus_{\mathscr{S} \times \mathscr{T}})$*.* Consider a $\mathscr{S} \times \mathscr{T}$-semiring.
$\{\langle ab, 5 \rangle, \langle a, 3 \rangle\} \oplus_{\mathscr{S} \times \mathscr{T}} \{\langle ab, 2 \rangle, \langle b, 1 \rangle\} = \{\langle ab, 2\ min\ 5 \rangle, \langle a, 3 \rangle, \langle b, 1 \rangle\} =$
$\{\langle ab, 2 \rangle, \langle a, 3 \rangle, \langle b, 1 \rangle\}$

Theorem 1 ($\overrightarrow{\mathscr{S}}$ is an idempotent semiring).

Nevertheless, in practice, solution 1 above is pursued, in which a simpler ranked $\mathscr{S} \times \overrightarrow{\mathscr{K}}$-semiring is used (as opposed to $2^{\mathscr{S} \times \overrightarrow{\mathscr{K}}}$ in definition 6) and the problem of multiple string outputs is delegated to a p-subsequentiality mechanism.

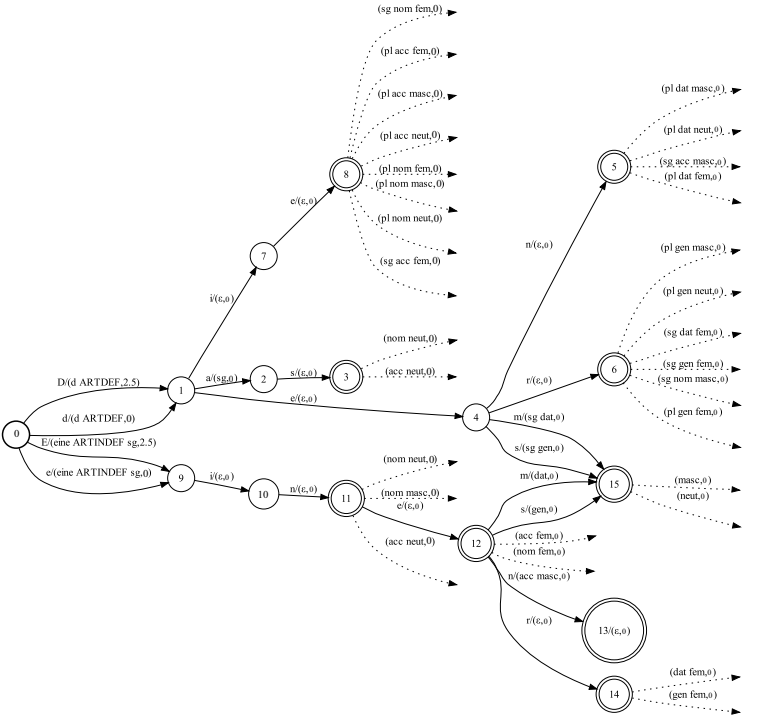

Fig. 2. 8-subsequential $\mathscr{S} \times \mathscr{T}$-WFSA A_{det} for the German articles

Figure 2 shows an optimal 8-subsequential WFSA over a $\mathscr{S} \times \mathscr{T}$-semiring, again for the system of German articles. To demonstrate the effect of the tropical semiring even for non-complex words as articles, we associate an arbitrary penalty score of 2.5 for article word forms starting with an uppercase letter (remember that in German, every word at the beginning of a sentence starts with an uppercase letter).[5] The WFSA demonstrates nicely the effect of weight pushing: while following the transition labeled with *d* from 0 to 1, the complex weight $(d\ ARTDEF, 2.5)$ is immediately outputted, since all paths starting with this transition share this weight as a longest common prefix weight.

As can be already seen from this tiny example, the resulting *p*-subsequential WFSA is much smaller: while the LFST has 33 states, the WFSA has only 16.

4 Experiments

For the experiment, we choose a lexicon of 47,000 geographic names including names of cities, countries, rivers, etc. taken from the lexicon pool of the TAGH morphology [8]. Each lexicon entry includes besides the name class an attribute indicating the semantic class following the LexikoNet framework [9]. To this lexicon, a geographic name grammar was applied allowing optional prefixation of lexicon entries having specific semantic classes. For example, country and continent names may take prefixes indicating the cardinal direction: *Süd+Ost+Afrika* (*Southern Eastern Africa*) or *Nord+Deutschland* (*Northern Germany*). Other possible prefixes are for example *Ober* (*Upper*), *Unter* (*Lower*) and *Zentral* (*Central*). To distinguish lexicalised prefix compound names (for example *Ostberlin*) from spontaneous prefixation, the name grammar assigns each occurrence of the boundary symbol + a penalty score.

Name lexicon and grammar were compiled into two different types of automata: the first one is a weighted LFST over the tropical semiring, the second a 6-subsequential WFSA over a $\mathscr{S} \times \mathscr{T}$ semiring.

Table 1 shows the sizes of the resulting finite-state automata. Both automata were constructed within the FSM<2.0> framework ([10]) and optimised for speed: to the LFST, ε-normalisation, transducer determinisation and encoded minimisation was applied; the string WFSA was determinised and minimised. Both automata are acyclic and accept around 245,000 geographic name forms.

As can be seen from the table, by using the WFSA, the number of states is reduced by nearly 82% and the number of transitions by over 75%.

Table 1. Sizes of the two versions of the geographic name analyser

Automaton type	# states	# transitions	# final states	# input ε	# output ε
LFST over $\mathscr{T}$	318,050	455,341	1	99,496	1,895
WFSA over $\mathscr{S} \times \mathscr{T}$	57,675	110,158	4,304	-	-

[5] Multiple outputs at the final states are indicated by dashed arrows.

5 Conclusion and Further Work

We showed that the use of a complex weight structure in combination with weighted finite state automata with multiple outputs can drastically reduce the size of CMS components. Of course, using variable length strings as weights in a WFSA comes with a certain prize, since they typically involve dynamic memory and speed optimisations which increase the memory consumption.[6]

However, there is the possibility of further implementational optimisations. The string WFSA from the experiment has 117,030 occurrences of weights (associated with the transitions and the final states), among them are 39,513 occurrences of the trivial weight $\langle \varepsilon, 0 \rangle$. The number of distinct weights is 30,506. By storing these distinct weights in a central table and replacing each weight occurrence w with the index of w in the table, the memory consumption can be significantly reduced.

References

1. Koskenniemi, K.: Two-level morphology: A general computational model for word-form recognition and production. In: Proceedings of COLING 1984, Stanford University, California, pp. 178–181 (1984)
2. Mohri, M.: Finite-state transducers in language and speech processing. Computational Linguistics 23, 269–311 (1997)
3. Mohri, M.: Minimization algorithms for sequential transducers. Theoretical Computer Science 234, 177–201 (2000)
4. Kuich, W., Salomaa, A.: Semirings, Automata, Languages. EATCS Monographs on Theoretical Computer Science, vol. 5. Springer, Heidelberg (1986)
5. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading (1979)
6. Mohri, M.: Semiring Frameworks and Algorithms for Shortest-Distance Problems. Journal of Automata, Languages and Combinatorics 7, 321–350 (2002)
7. Volk, M.: Choosing the right lemma when analysing german nouns. In: Multilinguale Corpora: Codierung, Strukturierung, Analyse. 11. Jahrestagung der GLDV. Frankfurt, pp. 304–310 (1999)
8. Geyken, A., Hanneforth, T.: Tagh: A complete morphology for german based on weighted finite-state automata. In: Yli-Jyrä, A., Karttunen, L., Karhumäki, J. (eds.) FSMNLP 2005. LNCS (LNAI), vol. 4002, pp. 55–66. Springer, Heidelberg (2006)
9. Geyken, A., Schrader, N.: LexikoNet, a lexical database based on role and type hierarchies. In: Proceedings of LREC (2006)
10. Hanneforth, T.: FSM<> 2.0 – C++ library for manipulating (weighted) finite automata (2004), `http://www.ling.uni-potsdam.de/fsm`

[6] For example, C++ string implementations like the one used in STLPort (`http://stlport.sourceforge.net`) apply a *short string optimisation* where short strings (e.g., up to 16 characters) are held in a static buffer, thereby avoiding allocation of dynamic memory, but also wasting bytes for even shorter strings.

JSLIM – Computational Morphology in the Framework of the SLIM Theory of Language

Johannes Handl, Besim Kabashi, Thomas Proisl, and Carsten Weber

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Germanistik und Komparatistik
Professur für Computerlinguistik
Bismarckstr. 6, 91054 Erlangen
{jshandl,kabashi,tsproisl,cnweber}@linguistik.uni-erlangen.de

Abstract. JSLIM is a software system for writing grammars in accordance with the SLIM theory of language. Written in Java, it is designed to facilitate the coding of grammars for morphology as well as for syntax and semantics. This paper describes the system with a focus on morphology. We show how the system works, the evolution from previous versions, and how the rules for word form recognition can be used also for word form generation.[1] The first section starts with a basic description of the functionality of a Left Associative Grammar (LAG) and provides an algebraic definition of a JSLIM grammar. The second section deals with the new concepts of JSLIM in comparison with earlier implementations. The third section describes the format of the grammar files, i. e. of the lexicon, of the rules and of the variables. The fourth section broaches the subject of the reversibility of grammar rules with the aim of an automatic word form production without any additional rule system. We conclude with an outlook on current and future developments.

Introduction

The NLP system JSLIM is the latest in a sequences of implementations within the framework of the SLIM[2] theory of language, which was introduced in [2]. The theory models the cycle of natural language communication, consisting of the hearer mode, the think mode, and the speaker mode. Providing the basis for human-machine communication, it uses the data structure of flat (non-recursive) feature structures called proplets and the time-linear algorithm of Left Associative Grammar (LAG)[3].

[1] The aim of the paper is to give a gentle introduction to the way grammars are written in JSLIM. The more technical aspects of the system, i. e. its scalability, the time required for analysis and generation, an evaluation of the system based on corpora, and a comparison of JSLIM to other existing systems, are the topic of a forthcoming paper.

[2] SLIM is an acronym for **S**urface compositional **L**inear **I**nternal **M**atching, i. e. for the basic principles on which the theory is based [1, p. 30].

C. Mahlow and M. Piotrowski (Eds.): SFCM 2009, CCIS 41, pp. 10–27, 2009.

JSLIM builds on the experiences made in the earlier systems Malaga and JLAG.[3] The present implementation is designed to free the grammar writer from all unnecessary work, thus simplifying development and upscaling. This is achieved by a declarative syntax for writing the rules. Implicit category value checks render the explicit insertion/deletion of values by means of imperative statements obsolete.

Currently, rules can be applied at different levels of the grammar. This allows an easier encapsulation of paradigmatic morphologic phenomena, e. g. inflection. The lexicon entries are flat, i. e. the nesting of feature structures as in Malaga is not permitted, though nesting can be simulated by means of symbolic references. These changes allow the extensive use of templates which considerably reduce the size of the lexicon and which improve its performance, readability and maintainability. The declarative syntax for inflectional and derivational combinations is bidirectional in that it may be used not only for the analysis of inflectional and derivational forms, but also for their generation.

1 Foundations

This section briefly describes the fundamental principles on which the system is built. First we explain the way an LAG works and on which principles it is based. Then we give the algebraic definition of a JSLIM grammar, because it works slightly different from the definition of an LAG given in [2, pp. 186–187].

1.1 LAG and SLIM

> Grammar does not tell us how language must be constructed in order to fulfil its purpose, in order to have such-and-such an effect on human beings. It only describes and in no way explains the use of signs. [6, p. 138]

The purpose of a grammar is therefore not to understand language but to find means by which to describe it. One of the most evident properties of natural language is its linearity in time. Therefore, LA-grammar uses a strictly time-linear derivation order.[4] [3] describes LAG as a bottom-up left-associative parsing scheme as illustrated in figure 1. In each derivation step, the *sentence start*, i. e. the part of the input already parsed, is combined with the *next word*, which in the case of morphological analysis corresponds to the next allomorph of the input. The number of combination steps required to successfully parse an input therefore corresponds to the number of allomorphs of the input minus one.

[3] The system Malaga [4] is quite elaborate and allows the construction of a grammar. Written in C, it provides a language for grammar rules and a single lexicon with nested feature structures. However, the style of the code requires programming experience. Also, the complexity of the lexicon entries makes the grammar less readable and upscaling more difficult. The next system, called JLAG, was implemented by [5] as an attempt to apply the paradigm of object-oriented programming to NLP. Coding became even more difficult and it never left the prototype state. As in Malaga the main problem was that the grammar writer was burdened with too much work and too little work was done by the system.

[4] It is the topic of the SLIM theory of language to model a real understanding of language.

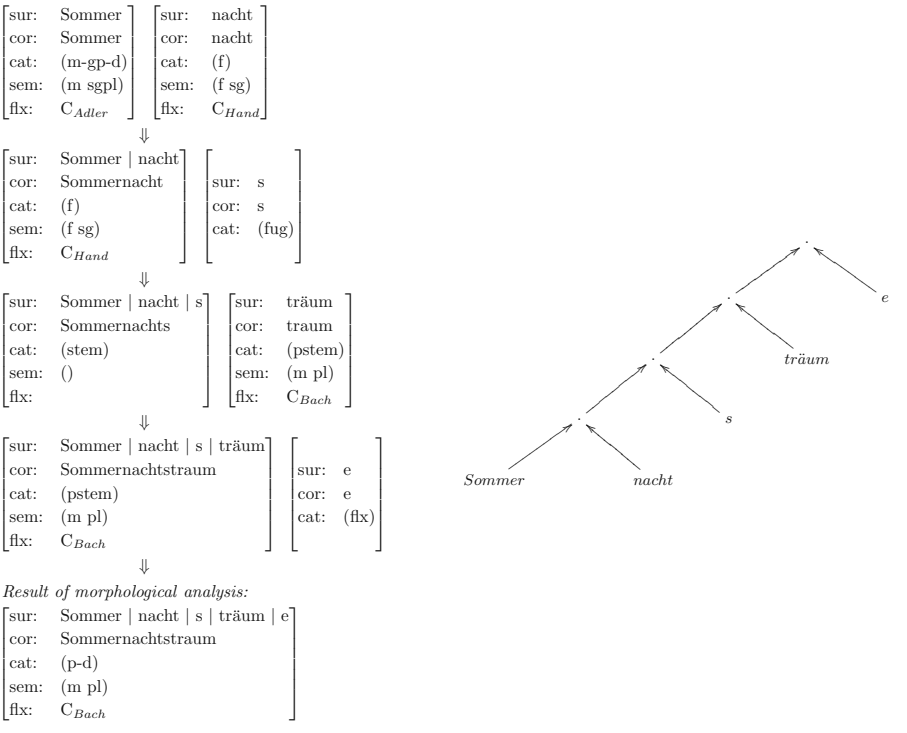

Fig. 1. The bottom-up left associative analysis and derivation order of an LA-grammar

1.2 The Algebraic Description of a JSLIM Grammar

According to [7, p. 4] a JSLIM grammar G' is defined as an 8-tuple

$$(W', A', C', LX', CO', RP', ST'_S, ST'_F)$$

Thereby

1. W' is a finite set of surfaces
2. A' is a finite set of attribute names
3. C' is a finite set of category segments
4. $LX' \subset W' \times (A' \times C'^*)^+$ is a finite set called the lexicon
5. $CO' = (co'_0, \ldots co'_{n-1})$ is a finite set of total recursive functions called categorical operations of the form

$$\mathscr{P}(FS) \times FS \rightarrow \mathscr{P}(FS) \tag{1}$$

Thereby, $FS \subset (A' \times C'^*)^+$ and $\mathscr{P}(FS)$ is the function to generate the power set for a given set. Let $co \in CO'$ and $co(\{fs_0, \ldots, fs_{n-1}\}, fs_n) \in \mathscr{P}(FS)$ with $fs_i \in FS$ for $i \leq 0 < n$. Let $FS_c = \{M \subseteq \mathscr{P}(FS) : |M| = c\}$. There exists an $m < n, k < n$ with

$$co\left(\bigcup_{i=0}^{k-1}\{fs_i\}\cup\bigcup_{i=k}^{n-1}\{fs_i\},fs_n\right)=\bigcup_{i=0}^{m-1}\{fs'_i\}\cup\bigcup_{i=k}^{n-1}\{fs_i\} \tag{2}$$

$$0\leq k<c_k=const \tag{3}$$

$$1\leq m\leq k+1 \tag{4}$$

$$\forall(M\in FS_{n-1-k})\ :\ co(\bigcup_{i=0}^{k-1}\{fs_i\}\cup M,fs_n)=\bigcup_{i=0}^{m-1}\{fs'_i\}\cup M \tag{5}$$

where $fs'_i\in FS$ for $i\leq 0<m$.

6. $RP'=(rp'_0\ldots rp'_{n-1})$ is a sequence of the same length with $rp'_i\subseteq\{i|0\leq i<n\}$, called rule packages
7. $ST'_S\subseteq(A'\times C'^*)^*\times RP'^*$ is a finite set called start states
8. $ST'_F\subseteq(A'\times C'^*)^*\times RP'^*$ is a finite set called final states

The definition is in accordance with the definition given in [2, p. 187]. However, feature structures are used instead of a category list to represent *sentence start* and *next word*. An LAG which works on feature structures was already defined in [8, pp. 37–38], but the concept of categorical operations has changed. One of the recent innovations regarding LAGs was to postulate a sentence start which is no longer coded in the form of a hierarchical data structure, but in the form of a set of flat feature structures. As a consequence, it is possible to access single values not only by using the underlying hierarchy of the data structure but also by using various – mainly syntactic-semantic – relations between the feature structures.[5]

Although those changes are fundamental as far as syntax is concerned they have little impact on morphological analysis where the sentence start is still coded into one single feature structure. I. e. in morphology we have the special case that both sets, *sentence start* and *resulting set*, always have a cardinality of one.

2 Applied Techniques

In this section, the newer concepts of JSLIM are presented. One of those concepts is the idea of *undirected programming*.[6] The idea is to specify rules as bijective functions, so that they can be executed in either direction. Here we only briefly cover the declarative syntax used in JSLIM, but we will go into more detail in section 4. We then investigate the techniques of *indirection*[7] and *common subtree sharing*, as they have an impact on

[5] This facilitates the way rules can be written and provides the grammar developer with new means of how to express constraints in the rules. [9] showed that rules for sentences with gapping and coordination can be modelled more accurately by exploiting word order.

[6] The most famous example of a programming language which allows undirected programming is Prolog. It is widely known among linguists for allowing easy coding of natural language grammars by using the definite clause grammar (DCG) notation. Although it is not the intention of the developers to create a new logic programming language, Prolog had a certain influence on the design of the rule syntax.

[7] Indirection has been widely used before, e. g. for constraint parsers, but never in combination with an LAG.

the storage and design of the lexicon.[8] The technique of *common subtree sharing* is also applied to the parsing process, e. g. for the internal representation of the parsing state.

2.1 Undirected Programming and Declarative Syntax

In JSLIM, a declarative syntax can be used to code inflection. Listing 1 illustrates the declension *table* of the German noun 'Bach' (creek) as it is coded in JSLIM. A table definition starts with the keyword `table` and the table name. The first letter of the table name must be an upper case letter. The table name is followed by a colon and the signature of the table. The signature defines, which attributes[9] of the combined feature structures are changed by one of the combination rules defined within the table body. The body comprises several rows with implication arrows. The left side of the arrow specifies the category values of *sentence start* and *next word*, whereas the right side specifies the resulting category values. The signature helps to investigate the attributes and the feature structure to which the values belong. To avoid the repetitive specification of the same category values for the *sentence start*, a semicolon can be put at the end of the last combination definition to indicate that the next definition will reuse the missing values from the current. A full stop, in contrast, indicates that the category values of the current definition are not reused.

```
table C_Bach: [cat,sem]      [sur] => [cat,sem]
              (m-g)  (m sg)   es   => (mg) (m sg)   ; # Baches
                             s    => (mg) (m sg)   ; # Bachs
                             e    => (md) (m sg)   . # Bache
              (pstem)(m pl)  e    => (p-d)(m pl)   . # Bäche
              (p-d)  (m pl)  n    => (pd) (m pl)   . # Bächen
```

Listing 1. A morphological rule in JSLIM

Hence, the above table definition defines the following combination steps illustrated in figure 2. This definition is declarative as it does not enforce by any means in what way the modifications have to be performed, but merely describes them.

2.2 Indirection

When developing a morphology for a natural language one of the first tasks of the developer is to somehow code the conjugation and declension tables. Although it is quite a simple task for an experienced traditional linguist to do so on a sheet of paper, it is not so clear at first sight how to perform this task within the framework of Left Associative

[8] The techniques of common subtree sharing, DAGs and suffix trees have been frequently used to reduce the size of lexica, inter alia in the field of chart parsers. The elegance of the approach of using templates as presented here is that the grammar developer benefits directly from the compact storage, as the continuous support through all the stages of parsing eases the building and maintenance of the lexicon.

[9] A description of the used attributes can be found in [1, p. 335]. The values of the attributes are coded using a distinctive (instead of an exhaustive) categorization [2, p. 244] [1, pp. 335–337].

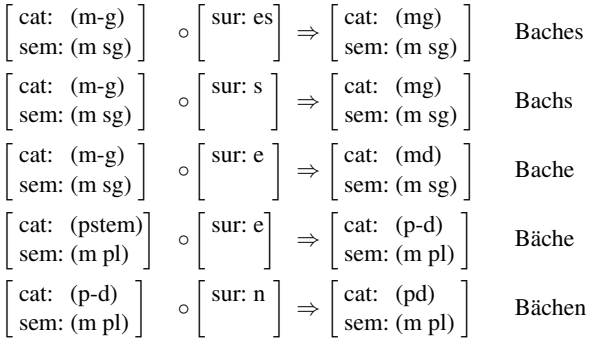

Fig. 2. A morphological rule in JSLIM

Grammar. An inflectional form is a combination of a stem with an inflectional affix. To restrict possible combinations of stems and flexives, the agreement conditions have to be coded into the category values of the combined parts. There are four possibilities to achieve this.

1. A *naive approach* would be to store a paradigm feature flag in the stem as well as in the inflectional suffix. This would reduce the inflectional check to a simple agreement check based on those two flags. The disadvantage of this approach is that, though agreement checks could be realized very efficiently, e. g. via a bitmap, the suffixes would become rather complicated, as suffixes are normally used in more than one paradigm, and difficult to code and to maintain. E. g. changing the paradigm feature of one paradigm also requires the paradigm feature flag of the inflectional endings of this paradigm to be changed which may inflict side-effects on other paradigms. Creating suffix entries for each paradigm is also no satisfactory solution, as this has a negative impact on the runtime.
2. The *stem approach* tries to restrict the combination of stem and inflectional affix mainly by focusing on the category of the stem. The disadvantage of this approach is obvious. As inflectional properties of the whole paradigm are coded into every single lexicon entry, storage is extremely redundant.
3. The *affix approach* tackles the problem from the other end. Instead of storing the inflectional properties of the paradigm in the stems, those properties are coded into the affixes. This avoids redundant storage of information but unnecessarily complicates the affix entries. A possible solution would be to multiply the lexicon entries of the affixes to simplify their categories. However, this has a noticeable negative impact on the run-time, as all these lexicon entries must be matched for a single combination with this affix.
4. Like the *stem approach*, the *indirection approach* accesses inflectional properties of the paradigm via the stem. However, the *indirection approach* is an improvement over the *stem approach* in so far as the information is only referenced, but not stored directly. The information itself is coded into a table, which is defined externally.

JSLIM supports all three approaches. However, the grammar developer is encouraged to use the *indirection approach*. In the next paragraph that approach is described in more detail.

$$\begin{bmatrix} \text{sur:} & \text{Bach} \\ \text{cat:} & \text{(m-g)} \\ \text{sem:} & \text{(m sg)} \\ \text{flx:} & C_{Bach} \end{bmatrix} \begin{bmatrix} \text{sur:} & \text{Bäch} \\ \text{cat:} & \text{(pstem)} \\ \text{sem:} & \text{(m pl)} \\ \text{flx:} & C_{Bach} \end{bmatrix}$$

Fig. 3. Allomorphs of the German noun 'Bach' (creek)

We code the allomorphs of a word as illustrated in figure 3. The lexicon entries contain an attribute for the surface (*sur*), attributes for the syntactic and semantic categories (*cat* and *sem*), and an attribute for inflection (*flx*). The value of the latter is a reference to the table defined earlier in listing 1. When the parser tries to combine the stem with an inflectional affix,[10] this trial is redirected to the table C_{Bach} which can be accessed via the attribute *flx*. Stem and inflectional affix become the so-called arguments of the table, i. e. a lookup in the table is performed, if an applicable combination rule can be found. In the case of a successful lookup the combination rule is applied, otherwise the table lookup fails and, as a consequence, the rule application also fails. Therefore, it is much easier to handle inflection by means of indirection, as matching is reduced to a simple test of whether the inflectional affix is allowed in the paradigm. No complex category checks are required.

2.3 Common Subtree Sharing

Reducing the memory needed by a system can be critical when trying to improve its performance. If a simple value requires much space and has to be stored multiple times, the easiest method to reduce the amount of memory needed is to store the value only once and then reference it. Though this may drastically reduce the amount of memory needed, still much memory is wasted for the pointers. Better results can be achieved by applying this principle of single storage to compound values. As illustrated in figure 4 and figure 5, most of the data structures used in JSLIM can be regarded as directed acyclic graphs. Those data structures bear a close resemblance to trees, with few nodes at the upper layers and almost all the nodes at the bottom layers. Besides, the co-domain of the values at the lower levels is quite restricted.[11] Therefore, the technique of common subtree sharing can be applied.

To allow the **sharing of common values in the lexicon**, values can be marked explicitly in the base lexicon and in the allo[12] lexicon. We will go into more detail in the next section. The notation reflects the way lexicon entries are stored internally. A lexicon entry is stored in the form of a 3-tuple which consists of the surface, the base form

[10] Inflectional affixes can easily be marked as such by an adequate category value.

[11] An exception are the values of the surface and base form attributes, which of course are specific.

[12] Following a tradition of former systems, JSLIM uses an allo lexicon and allo rules [10].

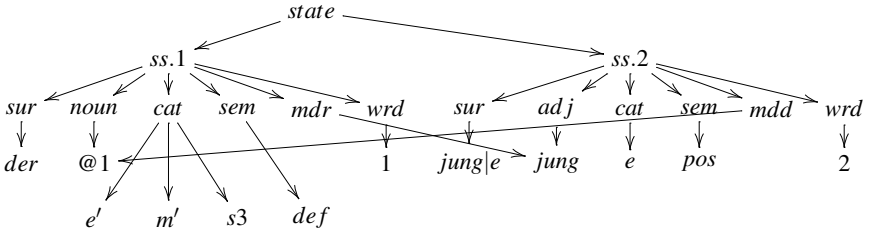

Fig. 4. Inclusion hierarchy of a parser, cf. [7, p. 55]

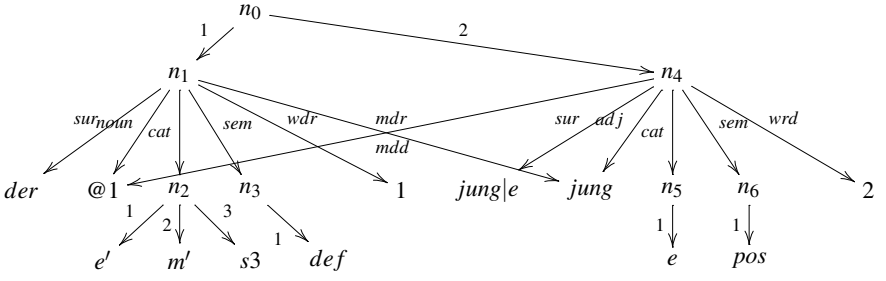

Fig. 5. Graph of a parsing state, cf. [7, p. 55]

sur :	der		sur :	jung | e		sur :	Mann		sur :	der		sur :	jung | e
noun :	@1		adj :	jung		noun :	Mann		noun :	Mann		adj :	jung
cat :	(e' m' s3)		cat :	(e)		cat :	(m-g)		cat :	(s3)		cat :	(e)
sem :	(def)		sem :	(pos)	+	sem :	(m sg)	→	sem :	(m sg def)		sem :	(pos)
mdr :	(jung)		mdd :	@1		mdr :	()		mdd :	(jung)		mdd :	Mann
wrd :	1		wrd :	2		wrd :	3		wrd :	1		wrd :	2

Fig. 6. Parser state when executing DET+N, cf. [7, p. 54]

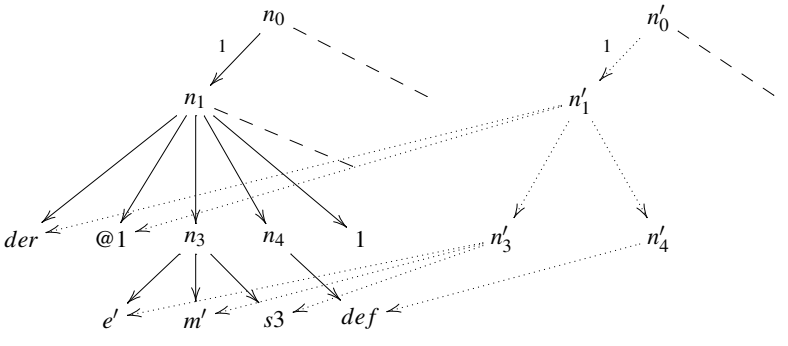

Fig. 7. Flat copy of a parser state, cf. [7, p. 56]

and a reference to a template which contains the additional values, normally shared with other entries.[13]

Sharing of common values of parser states is advisable, in so far as during a derivation ambiguity arises in (almost) every parsing algorithm from time to time. Ambiguities are normally handled by creating a new branch for each reading. The drawback of this approach is that it might not be possible to copy the temporary result of the derivation in constant time if it is coded in the current state. This, however, can be guaranteed if just flat copies are created and the depth of the graph is limited by a constant. However, the parser has to take care that no side effects are created. An example of copying a state in constant time is shown in figure 7. All values which will be modified by the action described in figure 6 are copied.

3 The JSLIM Grammar Files

In this section, the grammar files of a JSLIM grammar are briefly described. The focus lies on showing the differences to earlier and different implementations.[14]

3.1 The Lexicon

There are several ways of coding lexicon entries due to the following reasons:

- Depending on the state of a project the priorities may vary. While developing a grammar, the main focus might be the fast creation of a lexicon. Later on, it will probably shift to the readability, maintainability and space efficiency of the lexicon.
- Normally, a lexicon is not constructed by hand but by scripts which migrate existing lexica, or fill a lexicon with data extracted from corpora. Depending on the structure of the original data, conversion into one format might be easier to accomplish than into another possible format. And as long as the different formats are easily interchangeable, there is no reason why to restrict the lexicon to a single format.
- Normally, constructing a morphology component is a bottom-up process. First, a lexicon with a representative of each paradigm is needed to test inflection. Then, the lexicon is filled. Therefore it may be easiest, to start with a rather simple lexicon as long as it can also be automatically converted into a template based lexicon.

The different types of lexicon entries are described below.

Plain old lexicon entries are the simplest kind of lexicon entries. They are mainly used for lexicon prototyping. Their structure reminds us of the style used in [13]. All entries are coded separately. Although at first sight this might seem to be the best and easiest choice, plain old lexicon entries render the lexicon definition highly redundant as common values are not shared. An example for plain old lexicon entries is given in listing 2.

```
[sur: lern, cor: lernen, cat: (n' v), sem: (pres), ...]
```

Listing 2. A plain old lexicon entry for 'lernen' (to learn)

[13] By applying that technique, the amount of required memory can be considerably reduced, as was shown by the Italian morphology implemented by [11].

[14] A more detailed explanation of the grammar files can be found in [12].

The purpose of the **instance notation** is to avoid the above mentioned redundancy of categorical value. This aim is achieved with the help of templates. A template looks like an ordinary feature structure but is prefixed by the string `!template`. A feature structure can be marked as an instance of the last defined template by prefixing it with `!+`. A template followed by three instances is illustrated in listing 3. It is possible to override attribute values specified in the template by specifying them again in an instance. This style may be helpful for small to medium sized lexica, but only to a lesser extent for very large lexica.

```
!template[cat: (n' v), sem: (pres), ...]
!+[sur: lern,   cor: lernen]
!+[sur: erb,    cor: erben]
!+[sur: schenk, cor: schenken] ...
```

Listing 3. The instance notation for 'lernen', 'erben' (to inherit), 'schenken' (to make a gift)

The **sequence notation** can be used if feature structures differ in only one attribute value. Instead of specifying instances, the attribute to be added and the list of corresponding values are specified. For each value in the list an entry is added to the lexicon consisting of all the values specified by the last template in addition to the indicated attribute value pair. This style, however, is only possible if the entries belonging to a certain template differ in only one attribute value. This may be the case for a base form lexicon, but certainly not for the allo lexicon. In the latter, surface and base form attributes are word form specific and will differ in the majority of cases.

```
!template[flx: C_lernen, all: A_lernen]
!+sur cor: lernen erben ...
```

Listing 4. The sequence notation

A variant of the sequence notation is the **regexp notation**. It can be used in the case of instances differing in more than one attribute while all the attribute values are derivable from one value by means of regular expressions. This notation allows attribute names to be followed by regular expressions (see listing 5).

```
!template[cat: (n' a' v), sem: (pres), ...]
!+sur /(.+)en/\$1/ cor: lernen erben ...
```

Listing 5. The regexp notation

The **column notation** is the most compact one. It avoids the repeated declaration of attribute names and thus not only reduces the lexicon size but also increases processing speed: This kind of entry can be read in very efficiently as number and type of the attributes to be added is known in advance. Therefore, this is the preferred style for very large lexica.

```
!template[cat: (n' a' v), sem: (pres), ...]
![sur cor]
lern lernen
erb erben ...
```

Listing 6. The column notation

3.2 The Allomorph Method

Like the previous implementations of the LAG system (see [10, pp. 103–104]) JSLIM uses the allomorph method presented in [14, pp. 255–256]:

> The allomorph method uses two lexica, called the elementary lexicon and the allomorph lexicon, whereby the latter is automatically derived from the former by means of allo-rules before run-time. [...] During run-time, the allomorphs of the allomorph lexicon are available as precomputed, fully analyzed forms [...], providing the basis for a maximally simple segmentation: the unknown surface is matched from left to right with suitable allomorphs - without any reduction to morphemes.

Hence, the lexicon of the base forms coded in the above described notations merely serves as input for the allo rules to create the allo lexicon. Though, we will show in the next section, that the structure of the base form lexicon is preserved within the allo lexicon.

3.3 Allo Rules

Allo rules are coded in a declarative manner using the afore mentioned table notation (see section 2.1) and indirection. I. e. a reference to the allo rule is coded within the lexicon entry of the base form. Listing 7 shows the allo rule for the German noun 'Bach'. Allo rules are executed before run-time to create the allo lexicon [10]. The approach presented here is an improvement over earlier systems in which the allomorphs of a base form had to be generated by applying all allo rules.[15]

```
!template[cat: (n' a' v), sem: (pres), ...]
![sur cor]
lern lernen
erb erben ...
table A_Bach: [cor]                 => [sur,cor,cat,sem]
/(.*?)([AOUaou])(u?[^aouäöü]*)/ => /$0/ /$0/ (m-g) (m sg) ;
                                    => /$1$2"$3/ /$0/ (pstem) (m pl) .
```

Listing 7. Allo rule for the German noun 'Bach'

3.4 Allo Lexicon

In previous implementations the template structure of the lexicon files could not be maintained by the allo generator. The allo generator expanded the templates and the corresponding template instances. The output of the generator was a sequence of complete feature structures. Hence, templates were merely used to ease the coding of lexicon entries. This way of proceeding, however, is inefficient a) as far as the execution of the allo rules is concerned and b) with respect to further parser passes, e. g. the loading of the allo lexicon. An advantage of storing the allo table as an attribute value is that the allo rules which are needed can be called directly from the feature structure of the

[15] The generated allomorphs still contain a reference to the allo table. The purpose of this will be explained in section 4.

base form (indirection). It is therefore clear before run-time which allo rules have to be applied to which lexeme. What is more, the expansion of the templates can be avoided. The allo generator processes templates as follows: If a template contains an attribute *all*, the semantics of the allo rule is split into two parts: One affects the template and the other merely affects the template instance. The result of the generation therefore consists of one or more modified templates, one for every allo rule, each followed by a sequence of (probably) modified instances (see listing 8).

```
!+[all: A_Bach, flx: C_Bach]
!+cor: Aalkorb Abbrand Abbruch Abdampf ... Bach ...

                        ⇓ ⇓ ⇓

!+[all: A_Bach, flx: C_Bach, cat: (m-g), sem: (m sg)]
!+sur cor: Aalkorb Abbrand Abbruch Abdampf ... Bach ...

!+[all: A_Bach, flx: C_Bach, cat: (pstem), sem: (m pl)]
![sur cor]
Aalkörb Aalkorb
Abbränd Abbrand
Abbrüch Abbruch
Abdämpf Abdampf
...
Bäch Bach
...
```

Listing 8. Generation of the allo lexicon

3.5 Combi Rules

Combi rules define when two allomorphs can be combined. Normally, combi rules delegate most of the work to the tables which are referenced by the combined parts.[16] Figure 9 illustrates the rule used for inflection. A rule starts with its rule name, here STEM+FLX, followed by its rule package, i. e. by the set of follow-up rules. The *sentence start pattern* asserts that the *sentence start* is a stem. This is enforced by the declaration of the attribute *flx*. That the *next word* is an inflectional affix is ensured by the category value `(flx)` in the *next word pattern*. The value `*F` of the attribute `flx` triggers a table look-up with the feature structure which matches the sentence start and the feature structures which matches the next word as its argument.

```
STEM+FLX {STEM+FLX}

[cat: _, flx: *F] [cat: (flx)] => [...] [-]
```

Listing 9. Morphology rule for inflection

The rule is only applied, if the table look-up succeeds. The result of the look-up defines the way the two features structures are changed. The result patterns after the

[16] This is an example of indirection as described in section 2.2.

implication arrow can also be used to specify the way the two feature structures are changed. The pattern `[...]` indicates, that the features structure which matches the sentence start pattern remains unchanged within the sentence start. The pattern `[-]` means, that the feature structure which matches the next word pattern is excluded from the resulting sentence start.

3.6 Variables

For the patterns to become more abstract, variables can be used. For example, the rule in listing 9 contains the predefined anonymous variable _, which can be the placeholder for an arbitrary value. A variable can have a predefined data type and a co-domain as is illustrated in figure 10. The variable `CAT` has the data type `string` and may be bound to the values `{s1', s13', ...}`.[17]

```
string CAT <- {s1' s13' s2' s3' s3p2' p13' p2' m-g mg md ...}
```

Listing 10. Definition of the variable `CAT`

4 Word Form Generation

Word form generation deals with the creation of word forms from various inputs, be it a direct human request or the parameter values of the internal and external sensors of a robot. The application spectrum ranges from providing suggestions within a spell checker to the creation of a speaking robot. It is our goal to reuse the rules used for word form analysis for word form generation. This requires that the allowed combinations of lexicon entries are defined as a bijective function. The tables used in JSLIM can be seen as bijective functions in so far as they map a sequence of attribute values to another sequence of attribute values and are therefore reversible. If inflection is coded by means of tables, the code for the analysis can also be used for generation. As there is ongoing research in this field [15]. we cannot present a complete word form generation system with conceptualization and linearization. But we show that it is possible to generate a surface form for a given lemma and a given category – a process that is sometimes referred to as surface generation.

4.1 Surface Generation

We distinguish two cases of surface generation: a) the generation of a complete paradigm and b) the generation of a single word form. The first case may be of interest to check the correctness of a grammar, whereas the second case is needed for natural language production in the speaker mode.

[17] The grammar developer can also define constraints between pairs of variables. This feature is particularly expedient for specifying the agreement conditions in a syntax but is rarely used in morphology.

4.2 Generation of a Paradigm

To generate a complete paradigm for a base form, only an allomorph of the stem is needed.[18] To generate the complete paradigm the following steps are performed.

1. Look up the lexicon entry of the provided allomorph.
2. The allomorphy of the base form is specified in the attribute *all* of the lexicon entry, the value of which is a reference to a table. Apply that table to the allomorph. The circumstance that the rules in the table are applied to the allomorph and not to the base form, is negligible due to the fact that the application of the rules only depends on the base form of an entry. That base form is the value of the attribute *cor* and is the same for all allomorphs of a given stem. Besides, the category values are all changed by the applied rule, hence their particular values are insignificant.
3. Each allomorph has an attribute *flx* with an inflection table as its value. For each rule in the table, we look up the inflectional affix and execute the combination rule with the stem and the affix as its arguments. As it is possible to combine a stem with more than one inflectional affix, we repeat this step until no rule matches anymore.

We provide an example to illustrate the process of generating all word forms of a paradigm in more detail.

Example. Let ‘geb’ be the allomorph of a word form for which the paradigmatic forms shall be generated. The four allomorphs of the morpheme are ‘geb’ (Präsens), ‘gib’ (Präsens), ‘gab’ (Imperfekt) and ‘gäb’ (Konjunktiv 2). The first trivial step is to perform a lexical look-up for the allomorph ‘geb’, which returns the corresponding feature structure as illustrated in figure 8a.

Listing 11 shows the allo table which is used to create the allomorphs of the morpheme ‘geb’.

```
table A_geben: [cor] => [sur,cat,sem]
       /(.+)e(.+)en/ => /$1e$2/ (n' v)    (pres) ;   # geb
                     => /$1i$2/ (i' v)    (pres) ;   # gib
                     => /$1a$2/ (s13' v) (ipf)  ;   # gab
                     => /$1ä$2/ (n' v)    (k2)   .   # gäb
```

Listing 11. The table for the allomorphy of the German verb ‘geben’ (to give)

The application of the table is shown in figure 8b. The four allomorphs of the morpheme ‘geb’, namely ‘geb’, ‘gib’, ‘gab’ and ‘gäb’, are created on the basis of the allomorph ‘geb’. It is evident from the syntax of the rule that applying the table to different allomorphs of the same morpheme will not change the result, as the attribute values of the resulting feature structures are set by the respective applied rule. Listing 12 shows the inflection table of the generated allomorphs.[19]

[18] The stem morpheme is accessed via the surface of one of its allomorphs as an allo lexicon is used.

[19] Normally, only one inflection table is used for one paradigm. This, however, is solely a design decision and by no means obligatory.

a) Lookup b) Generation of the allomorphs

geb

$$\Downarrow$$

$$\begin{bmatrix} \text{sur: geb} \\ \text{cor: } \mathbf{geben} \\ \text{cat: (n' v)} \\ \text{sem: (pres)} \\ \text{all: } \mathbf{A\_geben} \\ \text{flx: C\_geben} \end{bmatrix} \begin{bmatrix} \text{sur: geb} \\ \text{cor: } \mathbf{geben} \\ \text{cat: (n' v)} \\ \text{sem: (pres)} \\ \text{all: } \mathbf{A\_geben} \\ \text{flx: C\_geben} \end{bmatrix} \Rightarrow \begin{bmatrix} \text{sur: geb} \\ \text{cor: geben} \\ \text{cat: (n' v)} \\ \text{sem: (pres)} \\ \text{all: } A_{geben} \\ \text{flx: } C_{geben} \end{bmatrix} \begin{bmatrix} \text{sur: gib} \\ \text{cor: geben} \\ \text{cat: (i' v)} \\ \text{sem: (pres)} \\ \text{all: } A_{geben} \\ \text{flx: } C_{geben} \end{bmatrix} \begin{bmatrix} \text{sur: gab} \\ \text{cor: geben} \\ \text{cat: } \mathbf{(n' v)} \\ \text{sem: } \mathbf{(ipf)} \\ \text{all: A\_geben} \\ \text{flx: } \mathbf{C\_geben} \end{bmatrix} \ldots$$

c) Execution of the combi rules

```
table C_geben:  [cat,sem] [sur] => [cat,sem]
                ...
                (i' v)(ipf) st => (s2' v)(ipf)
                ...
```

$$\begin{bmatrix} \text{sur: gab} \\ \text{cor: geben} \\ \text{cat: } \mathbf{(n' v)} \\ \text{sem: } \mathbf{(ipf)} \\ \text{all: A\_geben} \\ \text{flx: C\_geben} \end{bmatrix} \begin{bmatrix} \text{sur: } \mathbf{st} \\ \\ \\ \\ \\ \end{bmatrix} \Rightarrow \begin{bmatrix} \text{sur: gab|st} \\ \text{cor: geben} \\ \text{cat: } \mathbf{(s2' v)} \\ \text{sem: } \mathbf{(ipf)} \\ \text{all: A\_geben} \\ \text{flx: C\_geben} \end{bmatrix}$$

Fig. 8. Generation of a paradigm

```
table C_geben:  [cat,sem]        [sur] => [cat,sem]
                (n' v)  (pres)    e    => (s1' v) (pres) ;    # geb-e
                                  en   => (p13' v)       ;    # geb-en
                                  t    => (p2' v)        .    # geb-t
                (i' v)  (pres)    st   => (s2' v) (pres) ;    # gib-st
                                  t    => (s3' v)        .    # gib-t
                (s13' v)(ipf)     st   => (s2' v)        ;    # gab-st
                                  en   => (p13' v)(ipf)  ;    # gab-en
                                  t    => (p2' v)        .    # gab-t
                (n' v)  (k2)      e    => (s13' v)(k2)   ;    # gäb-e
                                  est  => (s2' v)        ;    # gäb-est
                                  en   => (p13' v)       ;    # gäb-en
                                  t    => (p2' v)        .    # gäb-t
```

Listing 12. The table for the conjugation of the verb 'geben'

For each generated allomorph the attribute values are matched with the left sides of the rules. In case of a match, we append the surface of the inflectional affix to the surface of the allomorph and perform the category changes defined by the rule.

4.3 Conditions for the Approach to Work

For surface generation to work, the following conditions must hold:

1. All categorical values are created by allo rules based on the value of the attribute of the base form.

2. The allo tables are stored as attribute values of the allomorphs. The alternative would be to read in the base form lexicon which would be impractical.
3. The inflection tables are stored as attribute values of the allomorphs. The alternative would be to try all inflection rules on a stem which would be impractical.
4. Allomorphy and inflection are coded only by means of tables. This is necessary as the tables allow different access orders as needed for generation.

These conditions must also hold for generating a single word form. However, they can easily be fulfilled without losing generative capacity and they rather facilitate lexicon design than hamper it. Hence, generation based on the rules for analysis is always possible.

Algorithm 1. generate($\mathscr{A}$,$\mathscr{F}$,v)

```
 1: for all a in 𝒜 do
 2:    if matches(rightside(a),v) then
 3:       return ⟨a⟩
 4:    end if
 5: end for
 6: for all f in ℱ do
 7:    if matches(rightside(f,v)) then
 8:       if ⊥ ≠ (r = generate(𝒜,ℱ,leftside(f))) then
 9:          return r + ⟨a⟩
10:       end if
11:    end if
12: end for
13: return ⊥
```

4.4 Generation of a Single Word Form

If all forms of a paradigm can be generated, it is possible to generate a single form by first generating the paradigm and then using a filter. The obvious disadvantage, however, is a lot of overhead. By using the following algorithm it is possible to use the tables to generate only a single word form of a given category:

1. Indicate the word form to be generated by providing an allomorph of the stem and one or more attribute values of the word form to create, e. g. `geb` and `[cat: (s2' v), sem: (ipf)]`.[20]
2. Look up the lexicon entry for the provided allomorph. This step is exactly the same as when generating a complete paradigm in figure 8a.
3. Determine the set of inflection rules for the allomorphs of the stem. If the allo rule does not change the inflection table (normal case), it can be taken directly from the lexicon entry. Otherwise, it can be investigated by looking at the right side of the rules of an allo table, as this is where they may be set. As the signature of the table `A_geben` in figure 11 does not contain the attribute `flx`, the referenced inflection table will not change when generating the allomorphs of the stem *geb* and can therefore be taken from an arbitrary allomorph.

[20] A filter can be used to allow a more flexible input which map more general user inputs to attribute values used in the paradigm.

4. For all allo rules, check if the result of one of these allo rules matches the provided category values. If it matches, the word form can be generated by applying the allo rule. E. g. if the provided attribute values were `[cat: (s13' v), sem: (ipf)]`, a matching form of *geben* could be created by executing the third row of the table specified in figure 11.
5. If the provided category values do not match any right hand side of an allo rule, check the inflection rules. If the right hand side of one of the inflection rules matches, the word form can be generated by applying that rule. Other inflection rules and at least one allo rule must be applied and the result of those applications must match the left side of the matching inflection rule. Therefore, we continue recursively with step 3, but use the values of the left side of the inflection table instead of the provided category values. E. g. let the specified attribute values of the verb *geben* be `[cat: (s2' v), sem: (ipf)]`. As these values do not occur on the right side of any rule in the table `A_geben`, the desired word form cannot be created by merely applying an allo rule. However, the values would match the right side of the fourth row in the table `C_geben` (cf. figure 12). Hence, we memorize that this rule has to be applied and try to find a way to generate the word form with the category values `[cat: (s13' v), sem: (ipf)]`, i. e. with the values of the left side of the applicable row.

Let $\mathscr{A}$ be the set of allo rules of the word form, $\mathscr{F}$ the set of inflection rules, and v the specified values. Let `leftside` be a function which returns the values of the left side of a combination rule except the next word, and `rightside` be the function which returns the right side of the combination rule. The algorithm is illustrated more formally in Alg. 1.

5 Conclusion

In this article, we presented the JSLIM system of automatic word form recognition and production. It has been shown how the current implementation uses techniques borrowed from computer science, e. g. common subtree sharing, undirected programming, and indirection. The combination of indirection and undirected programming in the form of tables seems to be an elegant approach for handling word form recognition as well as word form generation.

Currently, the morphological system is evaluated on the basis of a multitude of medium-scale grammars for different languages[21] and optimized for an increased performance. Further development will include extensive tests on corpora and a tighter integration of the morphology with the syntactic-semantic components.

References

1. Hausser, R.: A Computational Model of Natural Language Communication: Interpretation, Inference, and Production in Database Semantics. Springer, Heidelberg (2006)
2. Hausser, R.: Foundations of Computational Linguistics. Springer, Heidelberg (1999)

[21] This comprises the building of grammars with an average lexicon size of about 80000–100000 entries for languages like German, Italian, French and Polish.

3. Hausser, R.: Complexity in left-associative grammar. Theoretical Computer Science 106, 283–308 (1992)
4. Beutel, B.: Malaga 7.12. User's and Programmer's Manual. Technical report, Friedrich-Alexander-Universität Erlangen-Nürnberg (1995), http://home.arcor.de/bjoern-beutel/malaga/malaga.pdf (June 4, 2009)
5. Kycia, A.: Implementierung der Datenbanksemantik für die natürlichsprachliche Kommunikation. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (2004)
6. Wittgenstein, L.: Philosophical Investigations. Basil Blackwell Ltd., Oxford (1953)
7. Handl, J.: Entwicklung einer abstrakten Maschine zum Parsen von natürlicher Sprache. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (2008)
8. Schulze, M.: Ein sprachunabhängiger Ansatz zur Entwicklung deklarativer, robuster LA-Grammatiken mit einer exemplarischen Anwendung auf das Deutsche und das Englische. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (2004)
9. Kapfer, J.: Inkrementelles und oberflächenkompositionales Parsen von Koordinationselipsen. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (2009)
10. Schüller, G., Lorenz, O.: LA-Morph - Ein linksassoziatives Morphologiesystem. In: Linguistische Verifikation, pp. 103–119 (1994)
11. Weber, C.: Implementierung eines automatischen Wortformerkennungssystems für das Italienische mit dem Programm JSLIM. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (2007)
12. Weber, C., Handl, J., Kabashi, B., Proisl, T.: Eine erste Morphologie in JSLIM (in progress). Technical report, Friedrich-Alexander-Universität Erlangen-Nürnberg (2009), http://www.linguistik.uni-erlangen.de/clue/fileadmin/docs/jslim/morphology_docu.pdf (June 4, 2009)
13. Lorenz, O.: Automatische Wortformenerkennung für das Deutsche im Rahmen von Malaga. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (1996)
14. Hausser, R.: Modeling natural language communication in database semantics. In: Proceedings of the APCCM, vol. 96, Australian Computer Science Inc. CIPRIT (2009)
15. Kabashi, B.: Sprachproduktion im Rahmen der SLIM-Sprachtheorie. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (in progress, presumably 2009)

HFST Tools for Morphology – An Efficient Open-Source Package for Construction of Morphological Analyzers

Krister Lindén, Miikka Silfverberg, and Tommi Pirinen

Department of General Linguistics, University of Helsinki, Finland

Abstract. Morphological analysis of a wide range of languages can be implemented efficiently using finite-state transducer technologies. Over the last 30 years, a number of attempts have been made to create tools for computational morphologies. The two main competing approaches have been parallel vs. cascaded rule application. The parallel rule application was originally introduced by Koskenniemi [1] and implemented in tools like TWOLC and LEXC. Currently many applications of morphologies could use dictionaries encoding the a priori likelihoods of words and expressions as well as the likelihood of relations to other representations or languages. We have made the choice to create open-source tools and language descriptions in order to let as many as possible participate in the effort. The current article presents some of the main tools that we have created such as HFST-LEXC, HFST-TWOLC and HFST-COMPOSE-INTERSECT. We evaluate their efficiency in comparison to some similar tools and libraries. In particular, we evaluate them using several full-fledged morphological descriptions. Our tools compare well with similar open source tools, even if we still have some challenges ahead before we can catch up with the commercial tools. We demonstrate that for various reasons a parallel rule approach still seems to be more efficient than a cascaded rule approach when developing finite-state morphologies.

1 Introduction

Morphological analysis of a wide range of languages can be implemented efficiently using finite-state technologies based on finite-state transducers. Our goal is to implement efficient tools for creating and manipulating finite-state transducer morphologies for different uses and purposes. The task is daunting and we cannot do it alone.

Over the last 30 years, a number of attempts have been made to create tools for computational morphologies and some of them have withstood the test of time better than others. A major effort that has shaped the landscape and incorporated many lasting ideas is the morphological development tools created by Xerox. It started with the insight that we can use transducers to describe or encode phonological processes and relate various levels of linguistic abstraction using tools like TWOLC introduced by Koskenniemi and Karttunen [1,2,3]. To efficiently compile large-scale lexicons into transducers, we need special lexicon compilers like LEXC described by Karttunen [4,5].

Such tools do not solve all the problems. Writing full-scale dictionaries in LEXC may well be compared to having programmers write sophisticated applications in C without access to any of the modern high-level libraries. It is possible, but unless it is

C. Mahlow and M. Piotrowski (Eds.): SFCM 2009, CCIS 41, pp. 28–47, 2009.

done in some principled way, one may easily end up with spaghetti-code that is difficult to maintain. This is not the fault of the lexicon compiler, but the general programming solution is to create several descriptions that are small and independent, i.e. modular. With this insight and as computers became more powerful, the initial calculus that was conceived for abstract objects like automata and transducers in TWOLC and LEXC was expanded and migrated into the lexical programming environment XFST documented by Beesly and Karttunen [6], where smaller lexical modules for various purposes can be tailored and combined using finite-state calculus operations.

The previous effort is well-worth studying, but currently additional ideas have established themselves such as weighted transducers for modeling aspects of language that deal with preferences or trends rather than strict rules or on/off phenomena. Many applications of morphologies could use dictionaries encoding the a priori likelihoods of words and expressions as well as the likelihood of their relations to phonetic representations or their lexical relations to other words in the same language or in different languages. The efforts to explore weighted finite-state transducers for natural language processing are ongoing in information retrieval, speech processing and machine translation to name a few of the main application areas involved.

Since we do not pretend that we could develop all the morphologies for all the languages ourselves, or even all the aspects of the tools needed to develop these morphologies, we have made the choice to create open-source tools and language descriptions. We hope as many as possible will participate in the effort by developing the tools further for common needs and special purposes.

In addition to the open source tools, we also encourage the commercial use of the final transducers created by the tools by providing runtime software[1] that is free for commercial purposes. Eventually this will allow software applications simply to select the appropriate transducer in order to process a language correctly allowing the programmer to ignore special characteristics of individual languages.

Recently, a number of open-source finite-state processing environments have emerged, e.g. for unweighted transducers there are the *SFST–Stuttgart Finite-State Transducer Tools*[2] by Schmid [7], *foma: a finite-state machine toolkit and library*[3] by Huldén, etc., and for weighted transducers there are *Vaucanson*[4] by Lombardy et al. [8], *OpenFST Library*[5] by Allauzen et al. [9], etc. These are valuable contributions to the open source software that we can build on.

Our particular goal currently is in providing the basic facilities for efficiently developing, compiling and running morphologies with or without weights. To achieve our goal, we decided to create a unified API[6], which is capable of interfacing various weighted and unweighted finite-state transducer libraries allowing us to incorporate new libraries as needed. Currently, we have interfaces to SFST and OPENFST. On

[1] `https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstRuntimeInterface`
[2] `http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html`
[3] `http://foma.sourceforge.net/`
[4] `http://www.lrde.epita.fr/cgi-bin/twiki/view/Projects/Vaucanson`
[5] `http://www.openfst.org/`
[6] `http://www.ling.helsinki.fi/kieliteknologia/tutkimus/hfst/documentation.shtml`

top of the unified API, we created a set of basic tools[7], e.g. HFST-TWOLC, HFST-LEXC, HFST-COMPOSE-INTERSECT, HFST-TEST, HFST-LOOKUP, etc. With these tools, we created or used real full-fledged morphological descriptions of different languages from different language-families[8], e.g. *English, Finnish, French, Northern Sámi* and *Swedish*. We used the morphological descriptions for testing the functionality of the tools and for evaluating the performance of the different libraries through a unified interface on the morphological development and compilation tasks.

The current article presents some of the main tools that we have created: HFST-LEXC in section 2, HFST-TWOLC in section 3 and HFST-COMPOSE-INTERSECT in section 4. For each tool, we present the main theoretical underpinnings of the implementation and illustrate them with some examples. We highlight the main design decisions that influenced the efficiency of the implementation and how, if at all, our implementations differ from their namesakes. In section 5, we briefly present the morphological descriptions that we use for demonstrating and comparing the efficiency of the implementation. In section 6, we evaluate the performance of our tools for parallel-rule application and compare them with the performance of the *foma* LEXC compiler and the *Xerox* tools, as well as the cascaded rule compiler of SFST. In section 7, we discuss the test results and present some aspects of future research and development. In section 8, we draw the conclusions.

2 HFST-LEXC

A lexicon compiler is a program that reads sets of morphemes and their morphotactic combinations in order to create a finite-state transducer of a lexicon. This finite state transducer was called a *lexical transducer* by Karttunen [5]. The lexical transducer may be further adjusted with e.g. phonological rules. The example for our lexicon compiler is set by LEXC of Xerox [6]. In LEXC, morphemes are arranged into named sets called sub-lexicons. Each entry of a sub-lexicon is a pair of finite possibly empty strings[9] separated by ':' and associated with the name of a sub-lexicon called a *continuation class*.

Below, we highlight the main design decisions that influenced the efficiency of the implementation and the main theoretical underpinning of compiling a LEXC description into a finite-state transducer. Our morphology example outlines the nominal inflection of four Finnish nouns as shown in listing 1. This example is a highly simplified version of the actual morphology.

There are at least three time consuming parts of the HFST-LEXC compilation process. First the compiler needs to parse the strings representing the entry morphemes. Traditionally LEXC allows multiple characters in a single symbol. The problem of finding the optimal partition of a string when compiling it into a finite-state transducer is optimizing the *tokenization* algorithm. The tokenization is discussed in section 2.1. The set of entries in each sub-lexicon form a *union*. There are a few alternative strategies for creating unions, which are briefly outlined in section 2.2. The combining of sub-lexicons is described in section 2.3 on morphotax.

[7] `https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstHome`

[8] `http://www.ling.helsinki.fi/kieliteknologia/tutkimus/omor/index.shtml`

[9] Entries of regular expression form are not covered here to simplify the presentation, but a full definition of an entry in this formalism allows an entry to be a regular language.

```
Multichar_Symbols

+noun +1 +a +d +h +m +AV+ +AV- +AVA +AVD +AVH +AVM
+all +gen +ptv +sg ~A ~K ~P

LEXICON Root
    akku+noun+1+a:ak~Ku+AVA   N1b "battery";
    alku+noun+1+d:al~Ku+AVD   N1b "beginning";
    kumpu+noun+1+h:kum~Pu+AVH N1b "heap";
    kyky+noun+1+m:ky~Ky+AVM   N1b "capability";

LEXICON N1b
    NounSg ;
    NounPtvA ;


LEXICON NounPtvA
    +sg+ptv:~A+AV+  Ennd ;

LEXICON NounSg
    +sg+gen:n+AV-   Ennd ;
    +sg+all:l+AV-le Ennd ;
    :n+AV-          Compounding ;

LEXICON Compounding
    Root ;

LEXICON Ennd
    # ;
```

Listing 1. A simplified HFST-LexC lexicon for some Finnish nouns

2.1 Efficient Tokenizing of a Sub-lexicon Entry

Lexicon entries are tokenized using a simple left-to-right longest match tokenizer algorithm. The entry is tokenized by going through the entry string, position by position, and looking up the longest symbols available using a very simple greedy tokenizer. If the tokenizer is incremental, it memorizes new tokens as it parses the input assuming that multicharacter tokens have been declared in advance. An alternative, but less efficient, strategy is to determine all the tokens in a separate pass in order to compose the entry string with a tokenizer-transducer implementing a greedy left-to-right matching or some other strategy to achieve the desired partitionings.

2.2 Efficient Union of Sub-lexicon Entries

The finite-state form of a sub-lexicon is a union of entry transducers. Building a union of entry transducers is a relatively straight-forward process. However, iteratively taking the union of n entries with the n+1th entry is not ideal. A faster approach, given that all our entries are simple finite strings is to build the sub-lexicon transducer as one large prefix tree, *trie*. Each entry starts with a label of the sublexicon it belongs to and ends with a label of its continuation class.

2.3 Efficient Implementation of Morphotax over Sublexicons

The strategy for combining sub-lexicons can be described with operations on finite-state algebra using named auxiliary symbols with overgenerating combinations which

are filtered by composition to achieve legal combinations. This is further described in the next subsection. An optimized strategy for making the sub-lexicon combinations is to minimize the trie into an acyclic transducer, after which the sublexicon labels can be dropped by replacing the transitions of the entry final sublexicon labels with transitions to the target states of the entry initial sublexicon labels creating a possibly cyclic transducer.

Combining Sublexicons using Standard Finite-State Transducer Algebra. We assume all standard finite state operations to be known. For an introduction, see Beesly and Karttunen [6]. We use the following notation: $\cup$ is union, $\cap$ is intersection, $\circ$ is composition, juxtaposition is concatenation. Latin characters represent symbols of language and the ε symbol is used for a zero-length string. Capital Greek letters Σ, Γ represent subsets of an alphabet. We define $\Sigma = \{a, b, \ldots\}$ as a subset of the alphabet used for representing the morphophonology of the language in LEXC definitions. Γ is the alphabet of the *auxiliary* symbols used in our rules in the morphotax implementation. We assume that $\Sigma \cup \Gamma = \emptyset$. We use the symbol $J \in \Sigma$ for *joiners* to delimit and combine morphemes in our morphotax. A joiner for an entry with a continuation class named x is denoted as J_x and a joiner for a sub-lexicon named y is denoted as J_y.

We introduce the compilation of lexicons using the example-lexicon in listing 1.

A single entry in a sub-lexicon, i.e., a line of code in a LEXC file, is referred to as a morpheme denoted by $\mathscr{M}$. A morpheme can be a subset of the language $\Sigma^\star$ appended with the joiner of a continuation class (1).

$$\mathscr{M} = \Sigma^\star \, J \tag{1}$$

E.g. the LEXC string entry *akku:ak∼Ku+AVA* with a continuation class *N1b* becomes *a k k:∼K u:+AVA* ε*:*J_{N1b}.

A sub-lexicon $\mathscr{L}$ defined by (2) is a union of morphemes as specified in section 2.2.

$$\mathscr{L} = J \bigcup_{\mathscr{M}_x \in \mathscr{M}} (\, \mathscr{M}_x \,) \tag{2}$$

E.g. the lexicon named *Root* consisting of *akku* and *alku* with continuation class *N1b* becomes J_{Root} (*a k k u* J_{N1b} | *a l k u* J_{N1b}).

We create a filter $\mathscr{F}$ defined by (3) for legal morpheme combinations by pairing up adjacent joiners.

$$\mathscr{F} = \bigcup_{J_x \in J} (\, J_x \, J_x \,) \tag{3}$$

To account for the special starting lexicon and the special ending lexicon, we define $J_{Root} \in J$ and $J_\# \notin J$. The root lexicon can be used in continuation classes as a target, e.g. for the compounding mechanism, but the end lexicon is not available as a lexicon name, so it is not part of the regular morphotax. To accommodate this, we extend the filter definition to $\mathscr{F}'$ as in (4).

$$\mathscr{F}' = J_{Root} \, (\, \Sigma^\star \, \mathscr{F} \,)^\star \, \Sigma^\star \, J_\# \tag{4}$$

This allows us to create the final transducer $\mathscr{R}$ with only legal combinations of sublexicons by composition (5).

$$\mathscr{R} = \bigcup_{\mathscr{L}_x \in \mathscr{L}} (\mathscr{L}_x)^\star \circ \mathscr{F}' \tag{5}$$

E.g., for the sublexicons *Root*, *N1b*, *NounSg*, and *Ennd* in listing 1, and their entries *akku* and *+sg+all:lle*, we get the disjunction of lexicons $L^\star$, which we filter using $L^\star \circ F'$ as shown in figure 1.

$$\begin{aligned}
L^\star = \quad & (\; J_{Root}\; a\; k\; k\; u\; J_{N1b} \mid J_{N1b}\; J_{NounSg} \mid \\
& J_{NounSg}\; {+}sg{:}l\; {+}all{:}l\; \varepsilon{:}e\; J_{Ennd} \mid J_{Ennd}\; J_{\#} \;)^\star \\
F = \quad & J_{Root}\; J_{Root} \mid J_{N1b}\; J_{N1b} \mid J_{NounSg}\; J_{NounSg} \mid J_{Ennd}\; J_{Ennd} \\
L^\star \circ F' = \; & J_{Root}\; a\; k\; k\; u\; J_{N1b} \;\; J_{N1b}\; J_{NounSg} \\
& J_{NounSg}\; {+}sg{:}l\; {+}all{:}l\; \varepsilon{:}e\; J_{Ennd} \;\; J_{Ennd}\; J_{\#}
\end{aligned}$$

Fig. 1. Filtering a single path in HFST-LexC with a morphotax filter

Finally, all the symbols in Γ are removed. While this is trivial, it introduces some indeterminism in the final transducer, which would otherwise have been introduced by building direct epsilon arcs. Its influence on the performance is further detailed in section 6.

According to our experiments, attaching weights to each entry works without modification of the lexicon compilation method.

3 HFST-TwolC

Two-level rules are parallel constraints on symbol-pair strings governing the realizations of lexical word-forms as corresponding surface-strings. They were introduced by Koskenniemi [1] and have been used for modeling the phonology of numerous natural languages. HFST-TwolC is an accurate and efficient open-source two-level rule compiler. It compiles grammars of two-level rules into sets of finite-state transducers. The rules are represented as regular-expression operations closely resembling familiar phonological re-write rules both to appearance and semantics.

The most widely known two-level rule-compiler existing at the moment is the *Xerox Two-Level Rule Compiler* (later TwolC) presented by Karttunen el al. [3]. It is proprietary software, which imposes some limitations upon its use. The HFST-TwolC compiler has been designed to be an open-source substitute for the TwolC compiler and has a syntax and semantics very similar to those of the TwolC compiler. Hence existing two-level grammars, designed to compile under the TwolC compiler, require very few modifications to compile correctly under HFST-TwolC.

Besides being an open-source program, HFST-TwolC also has other benefits compared with the TwolC compiler. Resolution of rule-conflicts is an important part of compiling two-level grammars. We know of at least one instance, where the TwolC compiler resolves rule-conflicts in an incorrect way (see section 3.2). It also compiles

epenthesis rules in a way, which denies the grammar-writer the full expressive power of two-level rules (see section 3.2). In HFST-TWOLC we have been able to remedy these shortcomings by compiling the rules with the *generalized restriction-operation* (later *GR-operation*), presented by Yli-Jyrä and Koskenniemi [10]. It allows compilation of two-level rules in a uniform way and makes conflict-resolution easy to tackle, while still permitting efficient compilation.

In section 3.1, we demonstrate the syntax and semantics of a two-level grammar-file using a small example from Finnish morphology. The example grammar maps the lexical forms given by the example lexicon in listing 1, presented in section 2, into surface-forms.

It is not possible to demonstrate all features of HFST-TWOLC in this article, but we try to highlight the few most important differences to show that it is easy to migrate from the TWOLC compiler to HFST-TWOLC.

We use the GR-operation to compile the grammar-rules in HFST-TWOLC. In section 3.2, we explain how the different types of two-level rules are compiled. Rule-conflicts and their resolution are covered in section 3.2.

3.1 An Example Grammar

An input-file for HFST-TWOLC consists of five parts: *the Alphabet*, *the Rule-variables*, *the Sets*, *the Definitions* and *the Rules*. The file-format has been modeled on the format used by the TWOLC compiler, and all parts of the grammar are present also in the TWOLC compiler except for the part declaring rule-variables. There are a few other differences, as well, most of which we will mention below. A complete list of known differences can be found in the HFST-TWOLC documentation[10].

The Alphabet. The alphabet of a two-level grammar contains all lexical symbols specified in the HFST-LEXC grammar together with their possible surface realizations. In the example grammar in listing 2, the alphabet contains all letters used in Finnish words together with the vowel-harmony archphoneme `~A`, the gradation morphophonemes `~K` and `~P`, as well as, the gradation-markers `+AV+`, `+AV-`, `+AVA`, `+AVD`, `+AVH`, `+AVM`.

All symbols in the grammar may be arbitrary strings of UTF-8 characters, but characters like `+`, `~` or white-space, which bear special meanings for the compiler need to be escaped using the escape-character `%`.

The letters in the example-grammar of listing 2 always correspond to themselves on the surface. The gradation-markers always correspond to zero and the archphoneme `~A` and the morphophonemes `~K` and `~P` have various surface-realizations. E.g. `~A` is always realized as either `a` or `ä`.

Each valid pair of a lexical symbol and its surface-correspondence has to be listed in the alphabet. This differs from the TWOLC compiler, where pairs may be omitted from the alphabet, if they are identity-pairs or are already constrained by some rule. Forcing the grammar-writer to declare all symbol-pairs, may result in some extra work, but it also prevents the creation of inadvertent pairs.

[10] `https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstTwolC`

```
Alphabet

   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z Å Ä Ö
   a b c d e f g h i j k l m n o p q r s t u v w x y z å ä ö
   %+AV%+:0 %+AV%-:0 %+AVA:0 %+AVD:0 %+AVH:0 %+AVM:0
   %~A:a %~A:ä
   %~K:k %~K:0 %~K:v
   %~P:p %~P:m ;

Rule-variables

   Cm Cs Cw ;

Sets

   Gradations = %+AV%+ %+AV%- %+AVA %+AVD %+AVH %+AVM ;
   BackVowels = a o u A O U ;
   UpperCaseVowels   = A E I O U Y Å Ä Ö ;
   LowerCaseVowels = a e i o u y å ä ö ;

   Vowels = UpperCaseVowels LowerCaseVowels ;

Definitions

   AlphaSeq    = [ \Gradations: ]* ;
   NonVowelSeq = [ \:Vowels ]* ;

Rules

   "~K:0 Gradation"
   %~K:0 <=> _  AlphaSeq Gradations:0 AlphaSeq %+AV%-:0 ;

   "%~K:v and %~P:m Gradation"
   Cs:Cw <=> _ AlphaSeq Cm:0 AlphaSeq %+AV%-:0 ;
      where Cs in (   %~K   %~P )
            Cw in (     v     m )
            Cm in ( %+AVM %+AVH ) matched ;

   "Vowel Harmony"
   %~A:a <=> :BackVowels NonVowelSeq _ ;
```

Listing 2. An example HFST-TWOLC grammar governing the surface realizations of the forms presented in the example lexicon in listing 1

Declaring all symbol-pairs in HFST-TWOLC is mandatory, as we have not yet implemented an *other-symbol* like the one in Xerox TWOLC [3] using the HFST interface. Besides the grammar-formalism, this also affects the compile-time for rules, which becomes more dependent on the number of symbol-pairs in the grammar.

The Rule-Variables. Like the XEROX compiler, HFST-TWOLC supports defining a set of similar two-level rules using a rule-schema with variables. During the compilation of the grammar, each schema is compiled into actual two-level rules, by substituting the variables with the values specified for them. All rule-variables, which are used in the grammar, need to be declared in the Rule-variables section.

The Sets. It is often convenient to name some classes of symbols, which are used in many rules. E.g. the class `BackVowels` in the example-grammar in listing 2, which contains all vowel-segments used in the grammar. The sets in HFST-TWOLC and TWOLC are very similar constructs.

In HFST-TWOLC, the Cartesian product of sets, or a set and a symbol, is always limited to the set of symbol-pairs declared in the alphabet. E.g. the equivalent expressions `BackVowel:BackVowel` and `BackVowel` will only accept the pairs `a:a`, `o:o`, `u:u`, `A:A`, `O:O` and `U:U`. Although it is conceivable, that they would accept e.g. the pairs `a:U` and `A:O`, they will not, since the pairs have not been declared.

All sets have to be declared in the sets section of the grammar. Of the five sets we have defined in the example grammar, the first four are defined directly using a symbol sequence. The fifth set `Vowels` is defined as the union of the sets `SmallVowels` and `BigVowels`.

The Definitions. Like character-sets, also regular expressions may be stored under a name and accessed later using that name. Named regular expressions are called definitions and may be used freely in the rules. Sets and previous definitions can be used in the definition of a new definition. The definitions in HFST-TWOLC and TWOLC are identical.

The Rules. A two-level grammar constrains the surface-realizations of lexical forms. The constraints are given as two-level rules, whose joint effect determines the set of valid correspondences for each lexical form. Each of the rules governs one realization of a lexical symbol in a context given by a regular expression of pairs of a lexical and surface symbol.

The syntax and semantics of rules in HFST-TWOLC and TWOLC are very similar[11]. Except for *surface-restrictions* concerning epsilon, i.e. epenthesis rules, the rules also work the same way.

An example of a rule is the rule governing vowel-harmony in our example grammar

```
"Vowel Harmony"
%~A:a <=> :BackVowels NonVowelSeq _ ;
```

It states that the archphoneme `~A` has to be realized as `a`, if the surface-vowel immediately preceding it is a back-vowel. It also disallows the pair `~A:a` in all other contexts.

The rule accepts the first correspondence in table 1 since the vowel preceding `~A` is `y`, which is not a back-vowel. It disallows both of the latter correspondences. In the second correspondence `~A` is realized as `a`, even though the preceding surface-vowel is

[11] `http://www.xrce.xerox.com/competencies/content-analysis/fsCompiler/fssyntax.html`

Table 1. Symbol-pair correspondences for demonstrating the vowel-harmony rules

k y ~K y ~A	k y ~K y ~A	k u m ~P u ~A
k y k y ä	k y k y a	k u m p u ä

not a back-vowel. This violates the => direction of the rule. In the third correspondence, ~A is realized as ä, but the preceding surface-vowel is u, which is a back-vowel. This violates the <= direction of the rule.

HFST-TWOLC allows a set of rules to be defined using variables or by giving a set of rule-centers. E.g. the rule which defines the basic constraint of gradation in our example grammar is a rule with three variables: Cs, Cv and Cm.

```
"%~K:v and %~P:m Gradation"
Cs:Cw <=> _ AlphaSeq Cm:0 AlphaSeq %+AV%-:0 ;
   where Cs in (   %~K   %~P )
         Cw in (     v     m )
         Cm in ( %+AVM %+AVH ) matched ;
```

Listing 3. Rule defining the basic constraint of gradation

Like ordinary alphabet-symbols, variables may be used both in the center of a rule and in its contexts.

When a rule with variables is compiled, it is split into sub-rules. These are obtained by substituting real alphabet symbols for the variables. The possible values of variables are listed in the where-clause following the rule.

3.2 Compiling the Rules and Resolving Rule-Conflicts

HFST-TWOLC compiles two-level rules, given as regular expressions of pairs, into finite-state transducers. All two-level rules may be constructed from simple surface-requirements, context-restrictions and surface-prohibitions. The compilation reduces the two-sided rules and rules with variables into combinations of such simple constructions, or subrules.

After compilation, the subrules are intersected, so that finally equally many rule-transducers are produced as there were original two-level rules. Intersecting the subrules of the two-level grammar rules takes up a considerable portion of the compile-time of the grammar.

Compilation of the rules is preceded by a phase called conflict-resolution, which modifies rule-contexts in order to prevent harmful interactions between the rules. After conflict-resolution the modified rule-set may be compiled as usual.

We use the GR-operation of Yli-Jyrä and Koskenniemi [10] to compile rules. Both compiling rules and conflict-resolution is simplified using the operation.

The compilation in HFST-TWOLC differs from TWOLC when epenthesis rules are compiled. As Yli-Jyrä and Koskenniemi [10] point out, epenthesis rules may be compiled as any other surface-requirement rules using the GR-operation. This increases the expressive power of the two-level grammar as explained below.

A general restriction of the pair-alphabet Σ is an expression $W \overset{n\diamond}{\Rightarrow} W'$, where the precondition W and postcondition W' are unions of expressions of the form $V_1 \diamond V_2 \diamond \ldots \diamond V_n \subset \Sigma^*(\diamond \Sigma^*)^n$, where $\diamond \notin \Sigma$ is a special marker-symbol and each V_i is a regular language of the alphabet Σ. Such an expression is compiled into a regular expression using the GR-operation as in (6).

$$\Sigma^* - \text{delete}_\diamond(W - W') \tag{6}$$

The operation $\text{delete}_\diamond$ in (6) rewrites each marker-symbol $\diamond$ into epsilon and leaves all other symbols intact.

We do not need the full expressive power of the GR-operation. Instead we use a restricted version $W \overset{2\diamond}{\Rightarrow} W'$, which is limited to compiling rules with one center and a number of contexts with a right and a left part. Hence we operate on preconditions and postconditions with two diamonds, i.e. $W, W' \subseteq \Sigma^* \diamond \Sigma^* \diamond \Sigma^*$.

We discuss compiling one rule first and then conflict-resolution, although logically conflict-resolution is done first and then the rules are compiled. This is easier to explain, because conflict-resolution is highly dependent on the way the rules are compiled.

Compiling One Rule. Yli-Jyrä and Koskenniemi [10] explain how ordinary two-level rules can be compiled using the GR-operation. We use slight variations of the same methods.

Surface-requirement rules and context-restriction rules need to be compiled in different ways. Surface-prohibition rules can be compiled in a similar manner as surface-requirement rules and double-sided rules are compiled, by intersecting the two directions of the rule.

The general restriction corresponding to the context-restriction rule $a{:}b \Rightarrow \bigcup_{i=0}^{n} L_i \_ R_i$ is given by (7).

$$\Sigma^* \diamond a{:}b \diamond \Sigma^* \overset{2\diamond}{\Rightarrow} \bigcup_{i=0}^{n} L_i \diamond \Sigma^* \diamond R_i \tag{7}$$

The surface-requirement rule requires an auxiliary definition. We define the inverse projection $[x{:}]$ of the input-symbol x using (8). Here x may be any of the input-symbols of pairs in Σ, including epsilon.

$$[x{:}] = \{x{:}y \mid x{:}y \in \Sigma\} \tag{8}$$

The general restriction corresponding to the surface-requirement rule $a{:}b \Leftarrow \bigcup_{i=0}^{n} L_i \_ R_i$ is given by (9).

$$\left(\Sigma^* \diamond [a{:}] - a{:}b \diamond \Sigma^* \cap \bigcup_{i=0}^{n} L_i \diamond \Sigma^* \diamond R_i\right) \overset{2\diamond}{\Rightarrow} \emptyset \tag{9}$$

The general restriction corresponding to the surface-prohibition rule $a{:}b \Leftarrow \bigcup_{i=0}^{n} L_i \_ R_i$ is similar. It is given by (10).

$$\left(\Sigma^* \diamond a{:}b \diamond \Sigma^* \cap \bigcup_{i=0}^{n} L_i \diamond \Sigma^* \diamond R_i\right) \overset{2\diamond}{\Rightarrow} \emptyset \tag{10}$$

Using the GR-operation, epenthesis rules have the same semantics as other surface-requirement rules. The rule $0{:}a \Leftarrow b \_ b$ rejects the correspondences bb and $b0{:}cb$, but accepts $b0{:}ab$.

The TWOLC compiler compiles epenthesis rules in a different way than HFST-TWOLC. In TWOLC, the rule $0{:}a \Leftarrow b \_ b$ becomes equivalent to the expression $\Sigma^* - (\Sigma^* bb \Sigma^*)$, which means that bb is rejected, but $b0{:}cb$ is accepted, provided that the pair $0{:}c$ is declared in the alphabet Σ. This makes it impossible to interpret one epenthesis rule as a special case of another epenthesis rule.

E.g. we might want the pair $0{:}v$ between two vowels, but the pair $0{:}w$ between two like vowels. This can be expressed by the rules

$$0{:}v \Leftarrow \text{Vowel} \_ \text{Vowel} \ ; \text{ and } 0{:}w \Leftarrow V_x \_ V_x, V_x \in \text{Vowel} \ ;$$

In HFST-TWOLC conflict resolution modifies the context of the more general rule. A correspondence with $0{:}t$ between like vowels becomes disallowed, but a correspondence with $0{:}s$ between like vowels is allowed. In the TWOLC compiler this is not possible.

Resolving Rule-Conflicts. Rule-conflicts are situations where different rules require a lexical string to be realized in different ways. Since each correspondence of a lexical string and surface string needs to be accepted by all rules in a two-level grammar, such lexical strings are filtered by the grammar. Using the GR-operation to compile the rules allows separating the processes of conflict-resolution and rule-compilation. Previously, these may have been more entangled, which would explain, why the conflict resolution of the XEROX compiler sometimes works in an unexpected way (see example below).

Like TWOLC, HFST-TWOLC only handles two kinds of conflicts: right-arrow conflicts and left-arrow conflicts. Right-arrow conflicts occur between context-restrictions with the same center-pair. Left-arrow conflicts occur between surface-requirements with centers having the same lexical symbol, but different surface-symbols.

Consider the rules

$$a{:}b \Rightarrow x \_ \ ; \text{ and } a{:}b \Rightarrow y \_ \ ;$$

These are in right-arrow conflict with each other. Like Xerox TWOLC, HFST-TWOLC interprets both rules as permissions and replaces them with one rule, whose context is the union of the contexts of the conflicting rules. Joining the contexts is easy when the rules are compiled using the GR-operation.

A left-arrow conflict is resolvable exactly when one of the rule-contexts is a sub-context of the other. A trivial example of a resolvable left-arrow conflict is given by the rules

$$a{:}b \Leftarrow \{d, e\} \_ \ ; \text{ and } a{:}c \Leftarrow d \_ \ ;$$

Here the alphabet Σ consists of the pairs $a{:}b$, $a{:}c$, d and e. This is resolved by replacing the more general context with the difference of that context and the more specific context as given by (11), where we have written the contexts as generalized restriction contexts.

$$(\Sigma^* \{d, e\} \diamond \Sigma^* \diamond \Sigma^*) - (\Sigma^* d \diamond \Sigma^* \diamond \Sigma^*) = \Sigma^* e \diamond \Sigma^* \diamond \Sigma^* \tag{11}$$

This example does not compile as expected under TWOLC. Conflict-resolution results in a grammar, which rejects all lexical strings containing *a* or *e*.

Right-arrow conflict-resolution may result in large rule-contexts which may be time-consuming to determinize. Left-arrow conflict resolution requires testing all pairs of surface-restriction rules concerning the same lexical symbol. This means that the worst-case time-requirement is quadratic w.r.t. to the number of rules in the grammar.

4 HFST-COMPOSE-INTERSECT

A lexicon compiled using HFST-LEXC and a grammar of two-level rules compiled using HFST-TWOLC are combined using the program HFST-COMPOSE-INTERSECT. It is an implementation of the *intersecting composition* algorithm presented by Karttunen [5]. The result of the operation is equivalent to the composition of the lexicon-transducer with the intersection of the rule-transducers.

Karttunen [5] observed that the intersection of the rule-transducers alone may be extremely large and computing it may take a long time, whereas intersecting composition allows the lexicon to restrict the intersection of the rule-transducers. This reduces compilation time significantly.

Although computers have become considerably faster since 1994 and they have more memory, computing the intersection of the rule-transducers can still be very space-consuming. We intersected the rule-transducers of the two-level implementation of OMORFI[12], i.e. Pirinen's [11] morphological analyzer for Finnish. Without the intersecting composition, the rule intersection took eleven hours using the same machine we used for conducting our other performance-tests. Hence we believe that intersecting composition is still a necessary operation when developing full-scale two-level morphological analyzers.

5 Full-Scale Morphological Analyzers Using HFST Morphological Tools

We test the performance of the HFST tools by building three full-scale morphological analyzers of varying complexities for French, Finnish, and Northern Sámi. All of them highlight different aspects of the compilation process. To verify the correctness of the compilation results, we analyzed corpora using the lexical transducers.

The French analyzer was built from the existing morphological full-form lexicon Morphalou[13]. The lexicon was translated into the LEXC format and it contains some 550000 entries in a single lexicon. Each entry represents a word form and its analysis. We chose this lexicon for testing HFST-LEXC with a large number of real entries.

The Finnish analyzer has two implementations, i.e. the version using the SFST compiler format of OMORFI which is Pirinen's [11] original analyzer for Finnish, and a reformulated version using a LEXC lexicon and a TWOLC grammar format. The reformulation was done manually by converting the morpheme sets of the original code

[12] `http://kitwiki.csc.fi/kitwiki/Main/OMorFiHome`

[13] `http://www.cnrtl.fr/lexiques/morphalou/`

into LEXC sublexicons and rewriting the phonological rules from replace cascades into TWOLC rule sets. While care has been taken to ensure the similarity of the implementations, it should be noted that the versions are not totally equivalent. We still think they are close enough for a meaningful comparison of the two approaches.

The Northern Sámi analyzer is an original LEXC and TWOLC based morphological analyzer developed in the Divvun Project[14]. It is a full-fledged analyzer developed totally independently of the HFST project and it has both a large number of sublexicons and a large number of rules.

The characteristics of the analyzers of the three languages are summarized in table 2. The first three of the columns summarize the HFST-LEXC lexicons stating the numbers of sublexicons, lexicon-entries and symbols used in the lexicons. The remaining three columns summarize the HFST-TWOLC grammars. They state the numbers of symbol-pairs, rules and subrules in the double-sided rules and rules with variables. The example for French has no entries in the last three columns, since it has no two-level grammar.

Table 2. Some numbers characterizing the lexicons and two-level grammars we used for testing

	HFST-LEXC			HFST-TWOLC		
Language	Sublexicons	Entries	Symbols	Pairs	Rules	Subrules
French	1	553158	87	—	—	—
Finnish	213	94278	301	169	12	76
Northern Sámi	870	105503	428	313	105	555

6 Performance Evaluation

The goal of the performance evaluation is to see how far we still have to go before we reach industrial-strength performance. Additionally, we wish to see how the performance of the LEXC and TWOLC approach with parallel-rules compares to the approach with cascaded-rules. To achieve these goals, we compare HFST with some other open source tools and an industrial strength implementation by Xerox. By compiling the analyzers mentioned in the previous section, we can also collect performance figures on real full-fledged morphologies for identifying the most significant remaining bottle-necks in our tools.

The HFST-LEXC and HFST-TWOLC tools mimic many of the Xerox LEXC and TWOLC functionalities, so the input-files for the HFST tools require very small modifications in order to compile using the Xerox tools, and vice versa. This makes it is easy to compare the performance of the HFST tools with the Xerox versions.

As the Finnish OMORFI analyzer has two almost identical versions: one using replace-rules for the SFST compiler and one using two-level rules for the LEXC and TWOLC tools, we are able to compare the efficiency of the two approaches to building morphological analyzers. We can do this because both approaches are built on top of the same underlying SFST software library with tools specialized for each approach.

[14] `http://www.divvun.no`

Below, we have five tables summarizing the results of the performance tests. The first table 3 compares the total compile times of the analyzers using the HFST tools, the Xerox tools, the foma LEXC tool and the SFST compiler. For foma, only the compile-time for the analyzer of French is given, as foma only comes with a LEXC[15] interface. For the SFST compiler, only the compile-time for the Finnish lexicon is given, as our only implementation with cascaded rules is for Finnish.

Table 3. Total compile-times using HFST tools, foma LEXC, SFST compiler and Xerox tools to compile lexical transducers. Times are in seconds.

Language	HFST tools	foma LEXC	SFST compiler	Xerox tools
French	19.06 s	16.87 s	—	5.46 s
Finnish	14.44 s	—	1682.04 s	1.83 s
Northern Sámi	228.20 s	—	—	24.61 s

The following three tables 4, 5 and 6 give the HFST compile-times for the analyzers of Finnish, French and Northern Sámi. The times have been broken down into sub-phases of the compilation in order to see where the bottle-necks are. The phases are explained below the tables.

Table 4. HFST-LEXC performance broken into the different phases of the compilation process. Times are in seconds.

Language	1	2	3	4	Total
French	5.82 s	1.45 s	0.11 s	11.67 s	19.06 s
Finnish	1.08 s	0.27 s	0.32 s	6.50 s	8.17 s
Northern Sámi	1.02 s	0.44 s	2.27 s	15.94 s	19.67 s

1. The entry parsing and tokenization (cf. Sect 2.1)
2. Union of entries (cf. Sect 2.2)
3. Morphotactic filtering (cf. Sect 2.3)
4. Other phases (minimizing results, etc.)

Finally, table 7 gives an indication of the maximal memory consumption during the lexicon compilations using the HFST tools and the Xerox tools.

All tests were conducted on an Intel computer with a Xeon E5450 64 bit 3.00 GHz CPU and 64 GB of memory. For the HFST tools, the times were extracted using the C language `clock` function. For other tools, the GNU `time` command was used. In order to monitor the memory consumption, we used the GNU `top` command.

HFST has both a weighted and an unweighted implementation, but the current tests were performed using only the unweighted implementation of HFST, i.e. in practice we used the unweighted SFST library implementation of the HFST tools.

[15] Foma also has an XFST interface.

Table 5. HFST-TWOLC performance broken into the different phases of the compilation process. Times are in seconds.

Language	1	2	3	Total
Finnish	0.11 s	0.05 s	1.41 s	1.57 s
Northern Sámi	2.27 s	1.43 s	27.12 s	30.82 s

1. Reading the input-file and creating auxiliary data-structures. Compiling rule-contexts into transducers.
2. Identifying and resolving rule-conflicts.
3. Combining contexts and centers of single surface-requirements and context-restrictions. Intersecting subrules of rules with variables and double-sided rules, in order to form the final rule-transducers. Minimizing and storing the rule-transducers.

Table 6. HFST-COMPOSE-INTERSECT performance broken down into the different phases of the compilation process. Times are in seconds.

Language	1	2	3	4	Total
Finnish	0.19 s	2.80 s	1.05 s	0.65 s	4.70 s
Northern Sámi	2.18 s	154.14 s	21.01 s	0.38 s	177.71 s

1. Reading lexicon-transducer and rule-transducers.
2. Computing intersecting composition.
3. Determinizing and minimizing the result of the operation.
4. Storing the minimized result of the operation.

Table 7. Maximum space required using HFST and Xerox utilities to compile the transducers. Space indications are in megabytes (MB).

Language	HFST-LEXC	HFST-TWOLC	HFST-COMPOSE-INTERSECT
French	596 MB	—	—
Finish	181 MB	13 MB	48 MB
Northern Sámi	180 MB	291 MB	1090 MB (1.1 GB)
Language	**Xerox LEXC**	**Xerox TWOLC**	
French	85 MB	—	—
Finnish	28 MB	3 MB	—
Northern Sámi	13 MB	12 MB	—

7 Discussion and Future Research

In this section, we discuss the evaluation results and suggest some further lines of research and development of the tools that the evaluation figures seem to indicate. Comparing the total compilation times for HFST tools, Xerox tools, foma LEXC and SFST compiler, shows that HFST is still a magnitude slower than the Xerox tools. However,

HFST compares well with, foma, the other open-source tool. The decrease in compile-time for the Finnish lexicon, when parallel-rules are used, is considerable by improving performance with almost two magnitudes. Most importantly, using the HFST tools is sufficiently quick not to slow down the development of full-scale morphological analyzers. Even large morphological analyzers like the analyzers for French compile in less than a minute and Northern Sámi in less than ten minutes.

7.1 HFST-LexC Performance

Comparing the HFST-LEXC compilation times for French and Northern Sámi given in table 4, we see that the entry parsing and tokenization as well as the uniono of entries is almost linear. The lexicon for French has about five times as many entries as the lexicons for Finnish and for Northern Sámi. This is a result of the tokenization and trie-union described in section 2, which speeds up the building of sublexicons.

We also see that the number of sublexicons influences the morphotactic filtering in the HFST-LEXC compile-time. The lexicon for French only has one sub-lexicon, whereas the lexicon for Northern Sámi has 870 sub-lexicons and the Finnish lexicon is inbetween.

There is one main part that dominates the time consumption in table 4, i.e. the final determinization and minimization. The determinization and minimization of the final result consumes 60–80% of the compile-time of the lexicons. We use a standard algorithm which may have a more efficient implementation for cyclic structures in the competing software libraries.

7.2 HFST-TwolC Performance

Examining the HFST-TWOLC compile-times for Finnish and Northern Sámi shows, that the last phase, i.e. combining contexts and intersecting subrules, takes up approximately 90% of the compile-time. The compile-time for this phase depends heavily on the intersection, subtraction and determinization algorithms used when implementing the HFST API.

In HFST, we have not yet implemented an *other-symbol* like the one in the Xerox TWOLC presented by Karttunen [3], the rule-transducers encode a number of unnecessary transitions. This slows down intersection, difference and determinization among other operations. It is probably the single most important factor slowing down HFST-TWOLC. Like intersection, intersecting composition is also affected by the lack of an *other-symbol*, since intersecting composition is sensitive to the number of transitions in the rule-transducers.

7.3 Parallel Rules vs. Cascaded Rules

It is interesting to see, that the two-level HFST-LEXC and HFST-TWOLC approach to compiling the OMORFI analyzer for Finnish is so much more efficient than the cascade of replace-rules, which constitutes the SFST implementation of OMORFI. We know, that the difference lies in the approach to compiling the lexicon and the rules, as the unweighted HFST morphology tools ultimately perform their transducer operations using the SFST library, even if this happens through the HFST API.

One possible reason for the speed-up is that HFST-LexC and HFST-TwolC are more constrained environments than the SFST utility `fst-compiler` by Schmid [7], which is used for compiling the SFST version of the OMorFi analyzer. We suspect that the great liberty in constructing rules using SFST may tempt the user to indulge in unnecessarily unconstrained ways of expressing replacements with a very local area of application. This manifests itself among other things as an increased compile-time.

Converting the LexC and TwolC version of the Finnish lexicon from the SFST lexicon compiler files took approximately a week of manual work. While doing this, we were able to slightly modify and improve the rules in order to remove some of the incorrect readings that were coming through as analyses of the Finnish cascaded rule analyzer, which had not been corrected before. This also indicates that the parallel rule set may be easier to test and debug than the cascaded rules, but first and foremost it confirms the well-known effect that a reduced compile-time is very significant for the development process as it allows an increased number of test cycles during a fixed time-span.

7.4 The *Other-Symbol*

We have demonstrated, that the morphology tools HFST-LexC, HFST-TwolC and HFST-Intersect-Compose provide a realistic open-source alternative for constructing morphological analyzers in the two-level framework. Still, there is room for improvement, as the performance of the Xerox tools show.

Currently the performance of both HFST-TwolC and HFST-Intersect-Compose correlates strongly with the number of symbol pairs in the alphabet of the two-level grammar. A significant optimization for these HFST tools would be the introduction of an *other-symbol*, which can represent the class of pairs bearing no special meaning to a rule. Such a symbol would decrease the number of transitions in rule-transducers. In case the number of symbol-pairs of the alphabet is large, this has a significant impact on the performance of both HFST-TwolC and HFST-Intersect-Compose. In practice, the introduction of the *other-symbol* makes both tools insensitive to the number of symbols in the alphabet of the grammar. We believe, that this may help us achieve rule compile-times closer to those of Xerox.

7.5 Future Directions

In our future research, we intend to look at various aspects of and methods for integrating the creation and use of weighted transducers in morphologies. It is already possible to compile both weighted two-level lexicons and grammars using HFST tools. These can be combined into weighted lexical transducers using the weighted version of HFST-Intersect-Compose. It is also possible to adjoin meaningful weights to lexicon-entries in HFST-LexC. Currently HFST-TwolC only provides a way to compile weighted rules with zero-weights. However, even this small beginning allows us to combine weighted lexicons and two-level grammars using weighted intersecting composition. We are currently working on useful ways to attach weights to two-level rules with applications for weighted two-level grammars.

We were able to compare the performance of a cascaded rule approach with a parallel rule approach using the same underlying finite-state library. However, using our full-fledged morphologies, we could also compare different underlying finite-state libraries on real compilation tasks in order to compare different algorithms and their implementations. A future task, would be to compare the performance of e.g. the SFST library with that of OpenFST. Our preliminary evaluation results show that efficient and well-suited determinization and minimization algorithms have a significant impact on the real-world morphology compilation task.

8 Conclusions

We have chosen to create open-source tools and language descriptions in order to let as many as possible participate in the effort of providing morphological analyzers for the languages of the world. The current article present some of the main tools that we have created based on our unified API for finite-state libraries. The tools include HFST-LexC, HFST-TwolC and HFST-Compose-Intersect. We have evaluated the efficiency of the current implementations in comparison with some of the similar tools and libraries available using several full-fledged morphological descriptions. Our tools compare well with other similar open source tools, even if we still have some challenges ahead before we can catch up with the commercial tools. We demonstrate that for various reasons a parallel rule approach seems to be more efficient than the cascaded rule approach when developing finite-state morphologies.

Acknowledgments

We are grateful to the Finnish Academy and to the Finnish Ministry of Education for funding the project. We are also grateful to Erik Axelson for implementing the HFST interface and to Kimmo Koskenniemi and Anssi Yli-Jyrä for many fruitful discussions as well as to the anonymous reviewers for their comments.

References

1. Koskenniemi, K.: Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production. University of Helsinki, Department of General Linguistics (1983)
2. Karttunen, L., Koskenniemi, K., Kaplan, R.: A Compiler for Two-Level Phonological Rules. CSLI Publications (1987), `http://www2.parc.com/istl/members/karttune/publications/archive/twolcomp.pdf`
3. Karttunen, L.: Two-Level Rule Compiler, Technical Report ISTL-92-2, Xerox Palo Alto Research Center (1992), `http://www.xrce.xerox.com/competencies/content-analysis/fssoft/docs/twolc-92/twolc92.html`
4. Karttunen, L.: Finite-State Lexicon Compiler. Technical Report, ISTL-NLTT2993-04-02, Xerox Palo Alto Research Center, Palo Alto, California (1993)

5. Karttunen, L.: Constructing Lexical Transducers. In: The Proceedings of the 15th International Conference on Computational Linguistics COLING 1994, I, pp. 406–411 (1994)
6. Beesley, K., Karttunen, L.: Finite State Morphology. CSLI Publications, Stanford (2003), http://www.fsmbook.com
7. Schmid, H.: A programming language for finite state transducers. In: Yli-Jyrä, A., Karttunen, L., Karhumäki, J. (eds.) FSMNLP 2005. LNCS (LNAI), vol. 4002, pp. 308–309. Springer, Heidelberg (2006)
8. Lombardy, S., Régis-Gianas, Y., Sakharovitch, J.: Introducing Vaucanson. Theoretical Computer Science 328, 77–96 (2004)
9. Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., Mohri, M.: OpenFst: A general and efficient weighted finite-state transducer library. In: Holub, J., Žďárek, J. (eds.) CIAA 2007. LNCS, vol. 4783, pp. 11–23. Springer, Heidelberg (2007), http://www.openfst.org
10. Yli-Jyrä, A., Koskenniemi, K.: Compiling Generalized Two-Level Rules and Grammars. In: Salakoski, T., Ginter, F., Pyysalo, S., Pahikkala, T. (eds.) FinTAL 2006. LNCS (LNAI), vol. 4139, pp. 174–185. Springer, Heidelberg (2006)
11. Pirinen, T.: Suomen kielen äärellistilainen automaattinen morfologia avoimen lähdekoodin menetelmin. Master's thesis, Helsingin yliopisto (2008)

fsm2 – A Scripting Language for Creating Weighted Finite-State Morphologies

Thomas Hanneforth

Department for Linguistics, University of Potsdam, Germany

Abstract. The present article describes *fsm2*, a software program which can be used interactively or as a script interpreter to manipulate weighted finite-state automata with around 100 different commands. *fsm2* is based on FSM<2.0> – an efficient C++ template library to create and algebraically manipulate weighted automata. *fsm2* is particularly well suited to create morphological analysers on the basis of weighted automata.

1 Introduction

fsm2 is a simple XFST-style interpreter [1] for FSM<2.0>, an efficient C++ template library to create and algebraically manipulate weighted automata. Like XFST it is based on a stack machine, that is, most *fsm2*-commands manipulate finite-state automata on a stack. The notable differences to XFST are:

1. The regular expression syntax is adopted from AT&T's LexTools [2].
2. All finite-state machines are always (sometimes trivially) weighted with an algebraic weight structure called a semiring (see Section 2).
3. All higher-level operations in FSM<2.0> (on regular expressions, grammars, lexicons, etc.) are based on a symbol specification. So, the first step in the *fsm2*-interpreter will consist in almost every case in loading a symbol specification file (see Section 4.1).
4. FSM<2.0> and its interpreter *fsm2* are open source.

The article is organised as follows: Section 2 introduces some formal notions concerning weighted automata, while Section 3 focuses on some of the operations supported by *fsm2*. Section 4 shows fragments of *fsm2* code for creating a weighted morphological analyser for German nouns. A short conclusion is given in Sec. 5.

2 Formal Background

Finite-state automata in FSM<2.0> are weighted with an algebraic structure called a *semiring*.

C. Mahlow and M. Piotrowski (Eds.): SFCM 2009, CCIS 41, pp. 48–63, 2009.

Definition 1 (Semiring). *A structure $\mathscr{K} = \langle W, \oplus, \otimes, \overline{0}, \overline{1} \rangle$ is a semiring [3] if it fulfills the following conditions:*

1. *$\langle W, \oplus, \overline{0} \rangle$ is a commutative monoid with $\overline{0}$ as the identity element for $\oplus$.*
2. *$\langle W, \otimes, \overline{1} \rangle$ is a monoid with $\overline{1}$ as the identity element for $\otimes$.*
3. *$\otimes$ distributes over $\oplus$:*
   $\forall x, y, z \in W : x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$ (left distributivity)
   $\forall x, y, z \in W : (y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x)$ (right distributivity)
4. *$\overline{0}$ is an annihilator for $\otimes$: $\forall w \in W, w \otimes \overline{0} = \overline{0} \otimes w = \overline{0}$.*

In the following, we will identify a semiring $\mathscr{K}$ with its carrier set W.

Definition 2 (Semiring properties)
Let $\mathscr{K} = \langle W, \oplus, \otimes, \overline{0}, \overline{1} \rangle$ be a semiring.

- *$\mathscr{K}$ is called **idempotent** if $\forall a \in \mathscr{K} : a \oplus a = a$.*
- *$\mathscr{K}$ has the **path property** if $\forall a, b \in \mathscr{K} : a \oplus b = a$ or $a \oplus b = b$.*
- *$\mathscr{K}$ is called **commutative** if $\forall a, b \in \mathscr{K} : a \otimes b = b \otimes a$.*

With the definition of a semiring at hand, it is possible to define *weighted finite-state automata*:

Definition 3 (Weighted finite-state acceptor). *A **weighted finite-state acceptor** (henceforth WFSA, [4]) $A = \langle \Sigma, Q, q_0, F, E, \lambda, \rho \rangle$ over a semiring $\mathscr{K}$ is a 7-tuple with*

1. *Σ, the finite input alphabet,*
2. *Q, the finite set of states,*
3. *$q_0 \in Q$, the start state,*
4. *$F \subseteq Q$, the set of final states,*
5. *$E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \mathscr{K} \times Q$, the set of transitions,*
6. *$\lambda \in \mathscr{K}$, the initial weight and*
7. *$\rho : F \to \mathscr{K}$, the final weight function mapping final states to elements in $\mathscr{K}$.*

An extension to WFSAs are *weighted finite-state transducers* (WFST). Weighted finite-state transducers have a second alphabet Γ of output symbols and a slightly extended definition of the transition set: $E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times \mathscr{K} \times Q$.

The following definition functionally relates strings accepted (or generated) by a WFSA to semiring weights[1]:

Definition 4 (Weight associated with a string). *Given a WFSA A, let a* path *$\pi = t_1 t_2 \ldots t_k$ be a sequence of adjacent transitions. Let $\Pi(q, x, p)$ for $x \in \Sigma^*$ be the set of paths from state $q \in Q$ to state $p \in Q$ such that the concatenation of input symbols in each $\pi \in \Pi(q, x, p)$ equals x. Let $\omega(\pi)$ the $\otimes$-multiplication of all transitions weights in a path π: $\omega(\pi) = w[t_1] \otimes w[t_2] \otimes \ldots \otimes w[t_k]$. The **weight associated with an input string** $x \in \Sigma^*$ **wrt a WSFA** A – denoted by $[\![x]\!]_A$ – is computed by the following equation:*

[1] For weighted transducers, the definition is extended to a pair of strings $[\![\langle x, y \rangle]\!]$.

$$[\![x]\!]_A = \bigoplus_{\substack{q \in F, \\ \pi \in \Pi(q_0, x, q)}} \lambda \otimes \omega(\pi) \otimes \rho(q)$$

Definition 4 means that weights along an accepting path (leading to a final state) are combined by abstract multiplication. If there are several paths for some input string x – in this case the WFSA is called *ambiguous* – the weights of all those paths are combined additively. Since the $\oplus$-operation is required to be commutative, the order in which the paths are combined is not relevant. Section 3.1 will shortly discuss some of the most commonly used semirings and their applications.

3 Features of *fsm2*

This section shows some of the capabilities of *fsm2* and its underlying software library.

3.1 Semirings

Table 1 summarises the semirings currently predefined in *fsm2*.[2]

Table 1. Predefined semirings in *fsm2*

Semiring name	Symbol	Definition	Comments
tropical	$\mathscr{T}$	$\langle \mathbb{R}^+_{+\infty}, min, +, +\infty, 0 \rangle$	
real	$\mathscr{R}$	$\langle \mathbb{R}, +, \cdot, 0, 1 \rangle$	
log	$\mathscr{L}$	$\langle \mathbb{R}_\infty, \oplus_{log}, +, \infty, 0 \rangle$	$x \oplus_{log} y = -ln(e^{-x} + e^{-y})$
string	$\mathscr{S}$	$\langle \Gamma^* \cup \{s_\infty\}, \wedge, \cdot, s_\infty, \varepsilon \rangle$	$x \wedge y =$ *longest common prefix* of x and y
arctic	$\mathscr{A}$	$\langle \mathbb{R}_{-\infty}, max, +, -\infty, 0 \rangle$	
Viterbi	$\mathscr{V}$	$\langle [0,1], max, \cdot, 0, 1 \rangle$	
string×tropical	$\mathscr{S} \times \mathscr{T}$		
tropical×tropical	$\mathscr{T} \times \mathscr{T}$		

The standard semiring is the tropical semiring which is chosen for applications for which the notion of a *distance* is important. For example, it can be used for edit distance calculations and also for assigning penalties in morphological grammars to measure morphological complexity ([5]). Since in the tropical semiring, the $\oplus$-operation in definition 4 corresponds to the computation of a minimum, among a set of competing analyses the one with the minimal distance or the minimal penalty score is chosen.

The Viterbi and real semirings are of great importance for all kinds of HMM-style statistical language processing to compute forward/backward or maximum probabilities of input strings (see [6] for details).

[2] The actually available semirings depend on compilation options in the Makefile of *fsm2*.

The string semiring treats (output) strings as weights. Although the string semiring is not commutative and string weighted finite automata can thus not be combined with intersection and composition, the string semiring is very useful for compactly representing big finite-state transducers, for example, morphological analysers, especially when extended with the possibility of associating several output strings (analyses) with an input string[3] (see the next section for an example).

Finally, the tuple semiring $\mathscr{T} \times \mathscr{T}$ is a *ranked* semiring, that is, it is idempotent and has the path property and is thus suitable to allow *best path operations* ([4]) on WFSA with weights which are pairs of real numbers. The *fsm2*-framework allows (flat) weight tuples with up to 10 component semirings.

Semirings in *fsm2* are changed with the `semiring` command. Typing for example `semiring tsr_x_tsr` switches the system to the $\mathscr{T} \times \mathscr{T}$-semiring.

3.2 Weighted Operations

Algebraic Operations. The *fsm2*-interpreter supports the full set of algebraic operations defined on WFSAs and WFSTs (see [8]). Table 2 gives an overview of the *fsm2*-commands associated with these operations.

Table 2. Commands for algebraic operations in *fsm2*

Command	Meaning	Applies to
`union`	Union	WFSA, WFST
`concat`	Concatenation	WFSA, WFST
`star`	Star closure	WFSA, WFST
`plus`	Plus closure	WFSA, WFST
`complement`	Complementation	Unweighted FSA
`intersect`	Intersection	WFSA over commutative semirings
`compose`	Composition	WFST over commutative semirings
`crossproduct`	Cross product of two WFSAs	WFSA over commutative semirings
`difference`	Difference of a WFSA and a FSA	WFSA
`project`	1st, 2nd projection	WFST
`reverse`	Reversal	WFSA, WFST
`substitute`	Substitution of WFSAs into a WFSA	WFSA
`bestpath`	Find the best path	WFSA, WFST over idempotent semirings with the path property

Equivalence Transformations. Equivalence transformations are operations which change or optimise the topology of a weighted automaton without changing its weighted language or relation according to definition 4.

[3] See Hanneforth: *Using ranked semirings for representing morphology automata*, in this volume.

Table 3 summarises the equivalence transformations supported by *fsm2*.

Table 3. Commands for equivalence operations in *fsm2*

Command	Meaning	Applies to
`rmepsilon`	Removes $\varepsilon/\varepsilon : \varepsilon$-transitions	WFSA, WFST
`determinize`	Weighted acceptor/transducer determinisation, depending on the type of FSM	WFSA, WFST
`minimize`	Minimises WFSAs	WFSA
`synchronize`	Tries to synchronise input and output symbols	WFST
`epsnormalize`	Tries to move output-ε-transitions towards the final states to reduce nondeterministic computation	WFST
`optimize`	For WFSAs, ε-removal, weighted determinisation and minimisation; for WFSTs, ε-removal and encoded determinisation/minimisation	WFSA, WFST
`push weights`	Pushes weights towards the initial or final states	WFSA, WFST
`collect-weights`	Combines transitions with the same source & destination state and same labels by applying $\oplus$ to the transition weights	WFSA, WFST
`connect`	Removes inaccessible states and transitions with $\overline{0}$-weight	WFSA, WFST
`sort`	Sorts transitions by input/output label or weight	WFSA, WFST

Although not all finite-state transducers and weighted finite-state acceptors can be determinised (see [9]), the *encoded minimisation* technique implemented in the `optimize`-command can nevertheless be used to make the FSM faster and/or smaller. Encoded minimisation treats the symbol pairs of a transducer and the symbol-weight pair of a weighted acceptor as a symbol taken from a new alphabet to obtain a weighted or unweighted acceptor. This machine is then determinised and minimised and afterwards decoded back by using the original encoding mapping.

Recent versions of FSM<2.0> started to support WFSAs with multiple outputs for semirings not having the path property, for example, the string semiring or tuple semirings which have the string semiring as a component.

A special case of that type of WFSA are *p-subsequential* WFSAs, that is, WFSAs with a deterministic transition function and a final weight function ρ which is defined as follows:

$$\rho : F \mapsto (\Delta^*)^k, 1 \leq k \leq p. \tag{1}$$

That means that a final state q can emit up to p different semiring weights, for example strings in the case of a string semiring.

Figure 1 shows a 8-subsequential $\mathscr{S} \times \mathscr{T}$-WFSA for the German definite articles. Determinisation and minimisation of p-subsequential WFSAs is already fully supported in *fsm2*; ε-removal is currently restricted to WFSAs with a special topology.[4]

[4] This is caused by the inadequacy (to a certain extent) of the string semiring for that purpose.

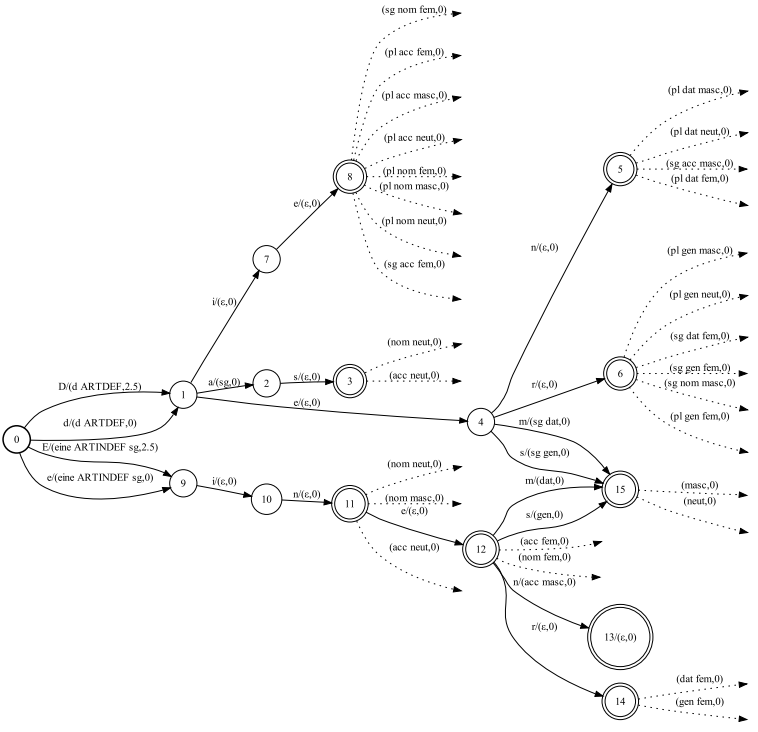

Fig. 1. p-subsequential $\mathscr{S} \times \mathscr{T}$-WFSA for the German articles

The symbolic part of the features of *fsm2* – that is, the capability of defining and compiling lexicons, grammars, replacement rules, etc. – is illustrated in the next section by means of an example taken from morphology processing.

4 Creating Morphological Analysers with *fsm2*

This section demonstrates the different symbolic formalisms integrated into *fsm2* by means of creating a weighted analyser for German nouns. The method used here is adapted from [5].

4.1 Symbol Signatures

Internally, all weighted automata work with 4-byte-integers as input/output alphabets. To allow symbolic computation, a *symbol signature* puts symbols and their internal integer representation in an one-to-one-mapping. The format of *fsm2*'s signature files is largely borrowed from LexTools [2]. Currently, there are two basic types of definitions in a symbol signature:

1. Type – subtype definitions
2. Category definitions

Listing 1 shows a fragment of a symbol signature.

```
# Supertypes
Lowercase       a b c d e f g h i j k l m ...
Uppercase       A B C D E F G H I J K L M ...
Letter          Lowercase Uppercase
Boolean         yes no

# Morphosyntactic features
Case            acc dat gen nom
Gender          fem masc neut
Number          sg pl

# Morphological features
NICSg           ic_sg_none ic_sg1 ic_sg2 ic_sg3 ic_sg4 ...
NICPl           ic_pl_none ic_pl1 ic_pl2 ic_pl3 ic_pl4 ...
StemType        def flex deko
DecoActive      Boolean
Bound           Boolean
Boundary        # ~ = -

# Morphological categories
Category:       NSTEM Gender NICSg NICPl StemType DecoActive Bound
Category:       NINFL Number Case NICSg NICPl
Category:       NSUFFIX Gender

# Syntactic categories
Category:       NN Gender Number Case
```

Listing 1. Fragment of a symbol signature for German noun morphology

Besides some definitions of super types like `Letter` and `Case`, the fragment showed in listing 1 basically defines two kinds of categories: *morphological* categories like `NSTEM` on the one hand and *syntactic* ones like `NN` on the other. `NSTEM` describes a noun stem morphologically, that is, associates it with information concerning gender, inflectional classes, type of stem, etc. The definition of the category `NINFL` does something similar for nominal inflectional affixes. On the other hand, the category `NN` defines a syntactic class (here taken from the STTS tagset [10]). The purpose of a morphology system is to provide a mapping from combinations of morphological categories to syntactic ones.

Subtype definitions can be nested and may form an acyclic inheritance hierarchy. All symbols defined in the symbol signature, whether category, sub- or super type, may be used in symbolic expressions. Additionally, a number of special symbols like <*?*> (default symbol), <*phi*> (failure symbol), <*bos*> (beginning of string), and <*eos*> (end of string) are added automatically. Symbol signatures are loaded in *fsm2* with the `load symspec` command; *fsm2* supports signature files in both ASCII and UTF-8 format.

4.2 Lexicons

Lexicons are simply treated as implicit disjunctions of regular expressions. Listing 2 shows a lexicon for some German noun stems based on the symbol signature of listing 1.[5]

[5] The operator : denotes the cross product operation. The full set of regular expression operations can be found in Table 4 in the appendix.

```
(Haus : Häus) [NSTEM Gender=neut NICSg=ic_sg1 NICPl=ic_pl3 StemType=flex|deko Bound=no DecoActive=yes]
(Haus : Haus) [NSTEM Gender=neut NICSg=ic_sg1 NICPl=ic_pl3 StemType=def|deko Bound=no DecoActive=yes]
(Tür  : Tür)  [NSTEM Gender=fem NICSg=ic_sg4 NICPl=ic_pl2 Bound=no DecoActive=yes]
(Holz : Holz) [NSTEM Gender=neut NICSg=ic_sg1 NICPl=ic_pl3 StemType=def|deko Bound=no DecoActive=yes]
(Holz : Hölz) [NSTEM Gender=neut NICSg=ic_sg1 NICPl=ic_pl3 StemType=flex|deko Bound=no DecoActive=yes]
```

Listing 2. Lexicon for some German noun stems

The lexicon fragment in listing 2 defines three noun lemmas *Haus* (*house*), *Tür* (*door*), and *Holz* (*wood*). Since nouns in German show allomorphic variation like stem umlaut, *Haus*, and *Holz* give rise to the definition of the stems *Häus* and *Hölz*, respectively. The feature `StemType` associated with a stem indicates whether it can be used in the singular (`def`), plural (`flex`) or in compounds or derived words (`deko`). Note that there are no allomorphs for *Tür*, thus its `StemType` feature is left underspecified.

Figure 2 shows the lexicon after being compiled into a finite-state transducer.[6] Note that it would be possible to assign unigram probabilities taken from a corpus already to the stems in the lexicon along the lines of [11], [12] or [13].

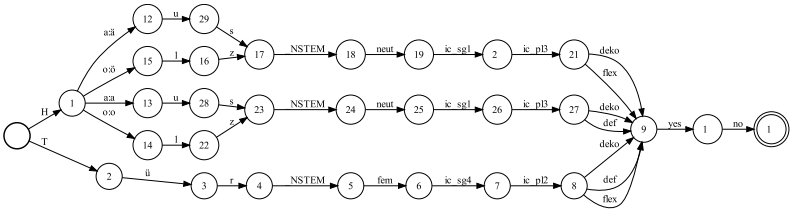

Fig. 2. Transducer for the noun lexicon fragment

For the nominal affixes, we create another lexicon where affixes are associated with morphosyntactic features and information about their inflectional classes. Listing 3 shows the affixes needed for the stems of listing 2. Note that some affixes have no overt form and consist solely out of feature specifications.

```
        [NINFL Number=sg Case=nom|acc|dat NICSg=ic_sg1]
s       [NINFL Number=sg Case=gen NICSg=ic_sg1]
        [NINFL Number=sg NICSg=ic_sg4]

en      [NINFL Number=pl NICPl=ic_pl2]
er      [NINFL Number=pl Case=nom|acc|gen NICPl=ic_pl3]
ern     [NINFL Number=pl Case=dat NICPl=ic_pl3]
        [NINFL Number=pl NICPl=ic_pl9]
```

Listing 3. Some inflectional affixes for noun stems

4.3 Grammars

The next step is to formulate constraints for the cooccurrence of noun stems and inflectional affixes: stems and affixes have to agree in the feature values of the inflectional classes (feature `NICSg` and `NICPl`). The traditional way – employed by XFST and its lexicon compiler *lexc* – is to divide all lexical material into equivalence classes and use

[6] Lexicons are processed with the `load lexicon` command.

the *follow class mechanism* to specify which classes can follow a given class. By following that approach, the linguist implicitly defines a finite-state automaton representing the allowed sequences of class members.

We think that encoding these kinds of automata manually is a tedious task. Instead, we believe that the more natural way of achieving the desired effect is to use a *grammar*. *fsm2* supports a restricted form of *weighted context-free grammar*, as described in [14][7].

Since in this section we follow the simple penalty score approach adopted from [5] to measure morphological complexity, we use a weighted grammar over the tropical semiring.

Listing 4 shows a fragment of a weighted grammar for German nouns.

```
[NOUN] ->        [FIRSTPART]* ([SIMPLENOUN]|[DERIVEDNOUN])

[FIRSTPART] ->  [STEM] [NSTEM DecoActive=yes StemType=deko] ([-]?) [#] <10>
[FIRSTPART] ->  ...

[SIMPLENOUN] -> [STEM] [NSTEM NICSg=ic_sg1 StemType=def Bound=no] \
                [AFFIX] [NINFL Number=sg NICSg=ic_sg1]
[SIMPLENOUN] -> [STEM] [NSTEM NICSg=ic_sg4 StemType=def Bound=no] \
                [AFFIX] [NINFL Number=sg NICSg=ic_sg4]

[SIMPLENOUN] -> [STEM] [NSTEM NICPl=ic_pl2 StemType=flex Bound=no] \
                [AFFIX] [NINFL Number=pl NICPl=ic_pl2]
[SIMPLENOUN] -> [STEM] [NSTEM NICPl=ic_pl3 StemType=flex Bound=no] \
                [AFFIX] [NINFL Number=pl NICPl=ic_pl3]


[DERIVEDNOUN] -> [STEM] [NSTEM StemType=deko DecoActive=yes Bound=no] \
                 [~] chen [NSUFFIX] <5> \
                 [AFFIX] [NINFL NICSg=ic_sg1 NICPl=ic_pl9]
[DERIVEDNOUN] -> ...
```

Listing 4. Fragment of a weighted grammar for nouns

The first grammar rule defines the basic form of a noun in German: zero or more occurrences of some first part ("*Erstglieder*") followed by a simple inflected noun or an inflected noun which is created by deriving other categories like verbs, adjectives, etc. The second rule says that an admissible first part of a noun may be another noun stem, followed by a strong morpheme boundary # (with an optional hyphen in between). This rule is weighted with an arbitrary penalty score of 10.[8] Of course, there are a lot of other possibilities here in German: the first part may be a derived noun, a verb, an adjective or a proper name.

The next four rules state the restrictions between noun stems and inflectional affixes by constraining stem and affix to the same inflection class.[9] Moreover, the plural rules require that the stem must have the value `flex` for the `StemType` feature, resulting in the choice of the correct allomorphic stem in the plural forms (for example, *Hölzern* instead

[7] Basically, a finite-state automaton cannot handle self-embedding.

[8] Again, some more sophisticated approaches based on the probabilities of the individual constructions may be adopted.

[9] Unfortunately, there is currently no variable-based agreement operation in *fsm2*. Introducing variables into grammar rules where the right-hand side can be an arbitrary regular expression (even with complementation or difference) is not an easy task.

of **Holzern*). The nonterminals `STEM` and `AFFIX` simply function as placeholders for the actual grapheme representations of the stems and affixes, resp.

The last rule for `DERIVEDNOUN` gives a example how to derive a noun with the diminutive suffix *chen*. Again, an arbitrary cost of 5 is assigned to this rule which also introduces a weak morpheme boundary $\sim$.

The grammar is then compiled into a WFSA with the `load grammar` command.

Basically, there are two ways to combine lexicons and grammars, which are both based on the corresponding closure properties of (weighted) regular languages and relations (see [15]):

1. Substitution
2. Intersection/composition

A (regular) *substitution* is a mapping from symbols of some alphabet Σ to a regular subset of Γ^* for some other alphabet Γ. In our grammar example, it would be possible to compile all grammar rules with the same left-hand side symbol into a WFSA and then substitute all categories and grammar nonterminals with the automata defined by lexicons or other grammar rules.[10]

We choose here to follow the second approach based on intersection. For that purpose, we first take the union of all lexical material; this encompasses the lexicons for stems and affixes, but also the morpheme boundaries. Afterwards, we take the closure of this union:

$$LexicalMaterial \equiv (NStems \cup NAffixes \cup MorphBoundaries \cup \ldots)^* \tag{2}$$

LexicalMaterial is then combined with the WFSA for the noun grammar – to which the two rules `[STEM] -> [Letter]+` and `[AFFIX] -> [Letter]*` were added – by composition:

$$NOUNS_0 \equiv LexicalMaterial \circ NounGrammar \tag{3}$$

4.4 Replacement Rules

Equations (2) and (3) already define the basic input/output relation of the morphological analyser. The next steps consists in *formatting* the input and output "tapes" of the WFST:

1. The input tape is restricted to letters. Some regular allomorphic processes like Schwa-insertion, etc. may be triggered by juxtaposing certain stems and affixes. Additionally, in compounds, word-internal noun stems must be written in lowercase letters.
2. On the output tape, the last occurrence of an instance of the categories `NSTEM` and `NSUFFIX` is mapped to the syntactic category `NN`. This formalises the *right-hand head rule* in German. All other pure morphological categories and features are deleted afterwards. Syntactic features like gender, number and case are not affected by the deletions

[10] In fact, the grammar compilation algorithm described in [14] works exactly in this way for the symbols defined within the same grammar.

Both tasks can be easily accomplished by using *replacement rules*. Listing 5 shows some of the rules for formatting the input tape of the analyser.

```
[Lowercase] => [Uppercase] / ([<bos>] | [-][#]) _

[Uppercase] => [Lowercase] / [#] _

[NSTEM]|[NINFL]|[NSUFFIX]|[Boundary] -> []
```

Listing 5. Input formatting

The first two rules are *parallel* replacement rules, that is, every upper or lowercase letter is replaced by its corresponding counterpart (the correspondence is defined by the order of the subtypes for a given super type in the symbol specification). The first rule says that every noun at the beginning of a word or after a word internal hyphen must be capitalised. The second rule says that all other word-internal occurrences of noun stems must be written in lowercase letters. The last rule is an *unconditional* replacement rule which deletes all categories and boundary symbols from the input tape.[11]

The rules for formatting the output tape are defined in a similar fashion[12] and are shown in listing 6.

```
[NSTEM] -> [] / _ (([-]?) [#]) / [~]

[Letter]+ -> [] / [NSTEM] _ [NINFL]

[Boolean]|[NICSg]|[NICPl]|[StemType]|[-]|[_NINFL] -> []

[_NSTEM]|[_NSUFFIX] -> [_NN]
```

Listing 6. Output formatting

The first rule in listing 6 deletes all `NSTEM`-categories which are followed by a morpheme boundary, while the second rule deletes all overt affixes occurring between `NSTEM`s and `NINFL`s. The third rule deletes all remaining morphological categories and features. Finally, the last rule rewrites the remaining instances of `NSTEM` and `NSUFFIX` to the category marker `NN`.

The two sets of formatting rules are compiled with the `load contextrules` command into finite-state transducers $FormatInput$ and $FormatOutput$, resp. Both are combined with the outcome of eq. (3) by (weighted) composition:

$$NOUNS \equiv (NOUNS_0 \circ FormatInput)^{-1} \circ FormatOutput \tag{4}$$

Eq. (4) first applies $FormatInput$ to $NOUNS_0$. Afterwards, the resulting WFST is inverted ($^{-1}$-operation) and then composed with $FormatOutput$.

4.5 Scripts and Functions

The commands for all the operations described above may be put in a *fsm2*-script, as depicted in listing 7.

[11] The rules modeling the allomorphic variation were omitted to keep the example simple.

[12] A complete list of the the (weighted) replacement rules defined in *fsm2* can be found in Table 5 in the appendix.

```
load symspec example
semiring tropical

load lexicon nlex
load lexicon ninfl
load lexicon nsuffix
regex "[Boundary]"
union 4
closure

load grammar ngram
compose

load contextrules format_input
compose
invert

load contextrules format_output
compose
optimize ilabel

lookup-file "words.txt" > results
```

Listing 7. Script constructing the WFST for nouns

The last command `lookup-file` takes a text file with a single word per line, looks up the word in the WFST and writes the results to another file.

The script language of *fsm2* also incorporates some simple *functions* (with or without parameters) to structure complex scripts.

Figure 3 shows the WFST which is the result of the script in listing 7.[13]

As expected, this WFST is cyclic, allowing arbitrary long nominal compounds. Since in the tropical semiring, the penalty scores reflecting the complexity of a word are added along a path, a compound like *Holzhaustürchens* (*wood house door dim.*) is mapped to a result string

$$\textit{Holz\#Haus\#Tür}\sim\textit{chen[NN Gender=neut Number=sg Case=gen]} \tag{5}$$

with weight $10+10+5=25$.

Since morphological analysers typically segment an input word in every possible way and may create a number of over-complex analyses, a subsequent n-best-path operation can to used to choose the n-best analyses.

5 Conclusion and Further Directions of *fsm2*

The present article introduced *fsm2*, a efficient scripting language for creating and manipulating weighted finite-state automata over different semirings. Since the underlying C++ framework FSM<2.0> has been developed for several years, every possible effort was made to make the operations as fast and robust as possible.[14] FSM<2.0> can handle weighted automata with several million states and transitions, the size of the automata

[13] Identity transitions of the form $a:a$ were collapsed to a single a-transitions. The trivial $\overline{1}$-weight of the tropical semiring (e.g., 0) has been removed from the transitions.

[14] Both FSM<2.0> and *fsm2* can be downloaded from `www.ling.uni-potsdam.de/fsm`

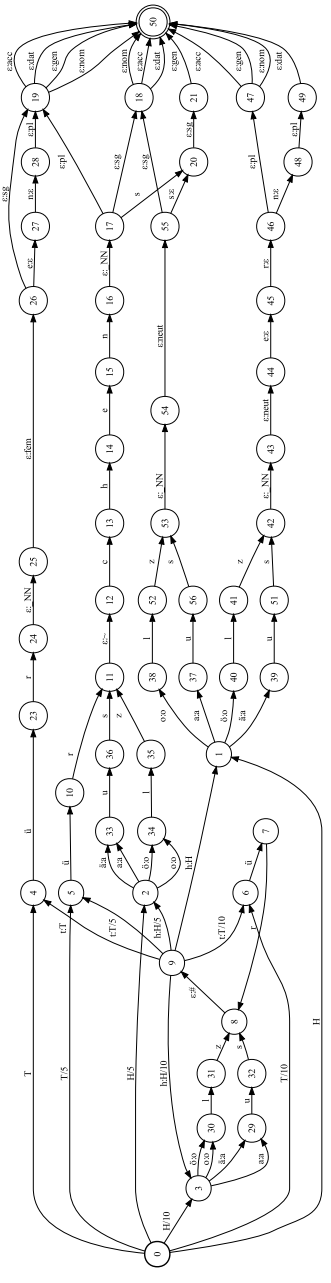

Fig. 3. The morphological analyser for nouns as a WFST

is only limited by the available memory. The compilers for the symbolic formalisms – lexicons, grammars and replacement rules – are also very efficient. The lexicon compiler compiles approximately 10,000 lines of a typical lexicon per second. Grammars

with several hundred rules are usually compiled in less than one second. The optimisation functions – ε-removal, determinisation, pushing and minimisation – are also very efficient and take even for big automata only a couple of seconds in the typical case.

Symbol specifications can be precompiled to support symbol sets with several hundred thousands symbols (useful for example for language-modeling tasks where every word constitutes an alphabet symbol).

The list of semirings in Table 1 can be expanded by providing a new C++ semiring class and registering it in *fsm2*.

Future versions of *fsm2* will include:

- More semirings, for example the *unification semiring* ([16]) and the *matrix semiring* ([15]).
- Commands supporting *language modeling*, for example creation of *backoff* ([17]) and *interpolation language models* ([18]), *smoothing*, *class based language models*, probabilistic taggers, etc.
- Commands for creating automata for pattern matching, for example, for suffix and failure transition automata ([19]).
- Support for p-subsequentiality for all algebraic operations.
- A fast incremental lexicon compiler for star-free lexicons creating already minimised automata ([20]).
- A mechanism for using variables within categories.
- Support for multi-core CPUs. This requires reworking nearly all algebraic operations to take advantage from 4 or 8 core CPUs which are now ubiquitous.

Acknowledgments. The author would like to thank the Department of Computational Linguistics at the University of Zurich, especially Manfred Klenner, Simon Clematide, and Michael Hess. The first version of *fsm2* evolved from an ERASMUS stay of the author at the institute in autumn 2007.

References

1. Beesley, K.R., Karttunen, L.: Finite State Morphology. CSLI, Stanford (2003)
2. Roark, B., Sproat, R.: Computational Approaches to Syntax and Morphology. Oxford University Press, Oxford (2007)
3. Kuich, W., Salomaa, A.: Semirings, Automata, Languages. EATCS Monographs on Theoretical Computer Science, vol. 5. Springer, Heidelberg (1986)
4. Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems. Journal of Automata, Languages and Combinatorics 7(3), 321–350 (2002)
5. Geyken, A., Hanneforth, T.: TAGH: A complete morphology for german based on weighted finite-state automata. In: Yli-Jyrä, A., Karttunen, L., Karhumäki, J. (eds.) FSMNLP 2005. LNCS (LNAI), vol. 4002, pp. 55–66. Springer, Heidelberg (2006)
6. Jurafsky, D., Martin, J.H.: Speech and Language Processing. Prentice Hall Series in Artificial Intelligence. Prentice Hall, Upper Saddle River (2000)
7. Hanneforth, T.: Using ranked semirings for representing morphology automata. In: Mahlow, C., Piotrowski, M. (eds.) Proceedings of SFCM. Springer, Heidelberg (to appear)
8. Mohri, M.: Weighted automata algorithms. In: Droste, M., Kuich, W., Vogler, H. (eds.) Handbook of Weighted Automata. Springer, Heidelberg (2009)

9. Mohri, M.: Finite-state transducers in language and speech processing. Computational Linguistics 23(2), 269–311 (1997)
10. Schiller, A., Teufel, S., Stöckert, C., Thielen, C.: Guidelines für das Tagging deutscher Textcorpora mit STTS. Technical report, Institut fur maschinelle Sprachverarbeitung, Stuttgart (1999)
11. Schiller, A.: German compound analysis with *fsc*. In: Yli-Jyrä, A., Karttunen, L., Karhumäki, J. (eds.) FSMNLP 2005. LNCS (LNAI), vol. 4002, pp. 239–246. Springer, Heidelberg (2006)
12. Junczys-Dowmunt, M.: Influence of accurate compound noun splitting on bilingual vocabulary extraction. In: Storrer, A., Geyken, A., Siebert, A., Würzner, K.M. (eds.) Selected Papers from the 9th Conference on Natural Language Processing KONVENS 2008, Berlin, Mouton de Gruyter, pp. 91–104. Mouton de Gruyter, Berlin (2008)
13. Lindén, K., Pirinen, T.: Weighted finite-state morphological analysis of Finnish compounding with hfst-lexc. In: Jokinen, K., Bick, E. (eds.) NODALIDA 2009 Conference Proceedings, pp. 89–95 (2009)
14. Mohri, M., Pereira, F.C.N.: Dynamic compilation of weighted context-free grammars. In: Proceedings of ACL 1998, pp. 891–897 (1998)
15. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading (1979)
16. Amtrup, J.W.: Efficient finite state unification morphology. In: COLING 2004: Proceedings of the 20th international conference on Computational Linguistics, Morristown, NJ, USA, vol. 453. Association for Computational Linguistics (2004)
17. Katz, S.M.: Estimation of probabilities from sparse data for the language model component of a speech recognizer. IEEE Transactions on Acoustics, Speech and Signal Processing 35(3), 400–401 (1987)
18. Jelinek, F.: Statistical Methods for Speech Recognition. In: Language, Speech and Communication. MIT Press, Cambridge (1997)
19. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Communications of the Asscociation for Computing Machinery 18(6), 333–340 (1975)
20. Daciuk, J., Watson, B.W., Mihov, S., Watson, R.E.: Incremental construction of minimal acyclic finite-state automata. Computational Linguistics 26(1), 3–16 (2000)

Appendix

Table 4. Built-in regular expression operators

Operator	Type	Meaning
`! RE`	Prefix	Complement
`$ RE`	Prefix	Contains operator
`RE *`	Postfix	Star closure
`RE +`	Postfix	Plus closure
`RE ?`	Postfix	Optionality
`RE /1`	Postfix	First projection
`RE /2`	Postfix	Second projection
`RE /r`	Postfix	Reversal
`RE /b`	Postfix	Best path
`RE /pw`	Postfix	Push weights towards initial state
`RE /pl`	Postfix	Push labels towards initial state
`RE /e`	Postfix	Remove epsilon transitions
`RE /d`	Postfix	Determinise
`RE /m`	Postfix	Minimise
`RE`	Infix	Operatorless concatenation
`RE | RE`	Infix	Disjunction
`RE & RE`	Infix	Intersection
`RE - RE`	Infix	Difference
`RE : RE`	Infix	Cross product
`RE @ RE`	Infix	Composition
`RE /i TYPE`	Infix	Ignore
`RE /iff RE`	Infix	Iff-suffix-then-prefix operator

Table 5. Replacement operators

Operator	Meaning
`alpha -> beta`	Obligatory replacement of an instance of `alpha` by an instance of `beta`
`alpha -> beta / gamma _ delta`	Obligatory replacement of an instance of `alpha` by an instance of `beta` if alpha is preceded by an instance of `gamma` and followed by an instance of `delta`
`alpha -> left ... right`	Obligatory bracketing of `alpha` with `left` and `right`
`alpha @-> beta`	Longest match replacement of `alpha` by `beta`
`alpha @-> left ... right`	Longest match bracketing of `alpha` with `left` and `right`
`supertype1 => supertype2`	Obligatory replacement of each direct subtype of `supertype1` with the corresponding direct subtype of `supertype2`

Morphisto: Service-Oriented Open Source Morphology for German

Andrea Zielinski[1], Christian Simon[2], and Tilman Wittl[3]

[1] FIZ Karlsruhe, Germany
andrea.zielinksi@fiz-karlsruhe.de
[2] Albert-Ludwigs-Universität Freiburg, Germany
simonc@informatik.uni-freiburg.de
[3] Ruprecht-Karls-Universität Heidelberg, Germany
wittl@cl.uni-heidelberg.de

Abstract. This paper presents the current activities surrounding Morphisto, an open source morphological analyzer/generator for German, based on the SMOR-based SFST tools. Morphisto was designed as a user-friendly application within the eHumanities project TextGrid. It comprises a *minimal* lexicon component that works in tandem with SMOR and is now being extended within the ELEXIKO project. Additional tools for the management of lexical data and services built on top of the finite state transducer are also integrated as Web Services in the grid, so that all resources can be shared readily among lexicographers, linguists, and finite-state developers.

Keywords: Computational German Morphology, Finite State Transducer, Grid, SFST.

1 Introduction

This paper aims at summarizing the main features of Morphisto, developed within TextGrid [1], a grid-based infrastructure for scholarly text-processing. It follows the goals and desiderata for a common research infrastructure for natural language processing based on finite-state technology in terms of generality, extensibility and availability [2]. As is well known, finite-state transducers are the first choice in computational morphology: They are declarative, can be used bidirectionally for analysis and generation, and are efficient in time and space. Our motivation for designing Morphisto was as follows:

- There are legacy restrictions which prevent all (at least known to us) morphological tools from open-source usage. This is especially true for the lexicon, which is the essential component of a morphological system for sophisticated real-life applications.
- There is a need for an easy-to-use tool that provides a user-friendly interface, particularly for non-professional users. The burden to install, update or even compile transducers on a local machine is not feasible in the context of eHumanities computing.

C. Mahlow and M. Piotrowski (Eds.): SFCM 2009, CCIS 41, pp. 64–75, 2009.

- For many NLP-applications, the lexicon has to be adapted. Therefore, an integrated data management program for easily extending the lexicon is essential.
- For many automatic annotation tasks, the quality of the morphological analysis is crucial. Therefore, additional components (i.e., functions for disambiguation and structural analysis) are integrated into the framework.

Outline: Initially, we will describe the general approach founded in SMOR and SFST as well as how we empirically evaluated the tool using a blind test. Secondly, we will present details about improving the Morphological Analyzer. In particular, we will show a recent extension of Morphisto, namely a disambiguation function that can be used to prune the number of analyses due to overgeneration. Thirdly, we will discuss how we adopted Morphisto for usage within real or research applications: TextGrid and elexiko. Within the elexiko project, the focus lies on establishing a gold-standard analysis for German morphology. Within TextGrid, however, the basic challenge is on the technical integration side.

2 Design of Morphisto

We based our work on the finite state toolkit SFST [3] that is distributed under an open-source compatible license. It builds on previous work done in DMOR [4] and DeKo [5] and comes with an elaborate set of rules for German inflection, compounding and derivation, as well as a minimal lexicon of approximately 1,000 example entries. Our aim was to cover the 30,000 most frequent words in the German language, and the DeReWo list[1] of lemmas, compiled from the DeReKo corpus[2] provided by the IDS Mannheim has been chosen as the training set.

The coverage of SMOR has been extended by (semi)automatic extraction of morphological information from the free edition published by the "Digitale Bibliothek"[3].

The word class statistics for the Morphisto lexicon is given in Table 1.

The construction of a large computational lexicon that can be freely distributed and that covers a large part of Standard Modern-German is described in [6], where also the standardized exchange format for the conversion to and from other finite state platforms is shown together with a database scheme.

3 Baseline: Test Results and Statistics

For our training set, we used the DeReWo list of 30,000 most frequent German words. The least frequent terms from this list have been seen <100 times in the COSMAS corpus. The Morphisto transducer lexicon is minimal in the sense that only those entries have been included that are needed to analyze our training set. The final (minimal) Morphisto transducer lexicon comprises approximately 18,200 entries, all of them manually

[1] `http://www.ids-mannheim.de/kl/derewo/` DeReWo List and User Documentation © IDS, Mannheim, 2007

[2] `http://www.ids-mannheim.de/kl/projekte/korpora/`

[3] Adelung – Grammatisch-kritisches Wörterbuch der Hochdeutschen Mundart, available from `http://www.zeno.org/Adelung-1793`

Table 1. Frequency of Morphological Units in the Transducer Lexicons

Lexicon	**Morphisto**
Basestems	17,339
– Nouns	7,833
– Proper Nouns	1,053
– Verb stems	4,300
– Adjectives	3,178
– Adverbs	781
– Closed Word Classes	190
Derivation Stems	67
Compound Stems	181
Prefix Stems	213
Suffix Stems (Derivation Rules)	410

checked. Some material from the IDS Fremdwörterbuch (dictionary of foreign words) has been integrated into the transducer lexicon and rule set. Most word formation affixes described in *grammis*[4] are now included in our rule set. Compiling a transducer lexicon with approx. 18,200 entries together with the grammar rules takes approximately 2:50 h compiled on a dual-core Pentium D 3.4 Ghz with 8 GB RAM running a Linux 2.6.16 kernel for x86_64 architecture.

The transducer performed best on our training set, the DeReWo lemma list, with a precision of almost 95%.

In our second test, we randomly selected subsets of 100 words in different frequency ranges from the *isowordlistde_DE file* provided by the open-source spelling checker ispell [7]. This list contains more than 225,833 German word forms unknown to Morphisto, including complex word formations. The test results are shown in Table 2. In this case, 159 out of 1,000 word forms (15.9%) could not be analyzed at all ('no result'). All in all, for 83.5% of the word forms a satisfactory analysis was provided by the transducer Morphisto, which is comparable to the results presented in [8].

More detailed results are given in Table 3, which also shows the number of correct, missing and spurious readings, e.g., the precision and recall rates.[5] Only for 6 out of 1,000 word forms the analysis was not correct (false positives). For instance, the German word 'allernächster' (very next), has been analyzed as NN-NN compound 'All' (universe) and 'Ler' (shade) (which is incorrect) followed by superlative of 'nah' (near) (which is correct). As a result of the prefix 'aller' (of all) not being stored in the lexicon, the correct analysis is missing.

Another source of error are ambiguous constituents which form part of a compound. For example, the compound noun *Rechtsempfinden* (sense of justice) could have been derived from the adjective modifier *rechts* (right) or the noun modifier *Recht* (law/justice).

[4] `http://hypermedia.ids-mannheim.de/pls/public/gramwb.ansicht`

[5] $Precision = \frac{Correct}{Correct+Spurious} * 100\%$ *and* $Recall = \frac{Correct}{Correct+Missing} * 100\%$.

Table 2. Test Results on Unknown ispell Words

Unknown Words	Total	Not Analyzed	Rate Analyzed (%)
F_0	1	0	100.00
F_3	5	0	100.00
F_5	23	1	95.67
F_7	62	6	90.32
F_9	179	25	86.03
F_{11}	179	22	87.71
F_{13}	179	9	94.97
F_{15}	179	11	93.85
F_{17}	179	24	86.59
F_{19}	179	22	87.71
F_{21}	179	31	82.68
F_{23}	179	35	80.45
F_{25}	179	39	78.21
$\sum$	1702	225	83.47 (Avg)

Table 3. Test Results on ispell

FC	Correct	Spurious	Missing	Precision (%)	Recall (%)
F_0	1	1	0	100.00	100.00
F_{16}	160	8	17	95.24	90.40
F_{17}	155	8	25	95.09	86.12
F_{18}	147	10	22	93.63	82.12
F_{19}	154	9	22	94.48	86.03
F_{20}	151	16	28	90.42	84.36
$\sum$	768	52	114	94.81 (Avg)	88.17 (Avg)

4 Improving the Morphological Analyzer

4.1 Handling Overgeneration: Which Is the Best Candidate?

For many applications (annotation, spell checking, etc.), it is desirable to automatically determine the most likely morphological analysis of a given word. Even without considering the context, humans often conceive only one reading for seemingly ambiguous words.

Currently Morphisto produces about 5 analyses on average. While derivation can, to some extent, be guided by grammatical rules, the process of compounding is almost unrestricted in German morphology. The large number of analyses is often due to the following kinds of errors: different segmentations (e.g., ‘West-europa’

(Western Europe) vs. 'Weste-ur-opa' (vest-ur-grandpa)), ambiguous constituents (e.g., der/die 'Chile-Kiefer' (jaw/pine)) or different levels of decomposition (e.g., 'Füller' (pen) vs. 'Füll-er' (fill-er)).

Consider the example 'Parkverbot'. Morphisto returns the following analyses:

(a) *parken<V>Verbot<+NN> (to park + interdiction)*
(b) *Park<NN>Verbot<+NN> (park/garden + interdiction)*

In our corpus, the combination *<V>Verbot<+NN>* is predominant. The distributional properties of language reveal that certain constituents are more often combined with one type of constituents than with others.

The following approaches have been followed for disambiguating morphological analyses: Schmid [9] presents an algorithm based on head-lexicalised probabilistic context-free grammars. The probabilities of the terminal symbols and production rules are estimated from a training corpus and a given CFG, using the inside outside algorithm, which is computationally very expensive. Geyken and Hanneforth [10] developed the tool TAGH based on weighted finite-state-automata. The probability of any path through the transducer is the product of the probabilities associated with each edge on the path, calculated as a cost function on each segmentation or derivation step. The more analysis steps the lower the overall probability.

Our approach is based on a Markov model that assigns a probability distribution over possible analyses in a postprocessing step, allowing for the ranking of analyses. We computed a bigram model out of a set of 10,000 manually selected valid analyzed words, randomly chosen from the elexiko word list. More precisely, our disambiguation algorithm has learnt the lexical and conditional wordclass probabilities (bigrams) from a manually annotated corpus of correct analyses, which contains information on the segmentation, lemma and PoS. All hypotheses are then scored and ranked according to the bigram model, using the Viterbi algorithm for calculating the probability of constituent sequences.

In figure 1, we show the disambiguation of '*Kochmütze*' (toque) based on a HMM. From the following two analyses

(a) *koch<V>Mütze<+NN>* and
(b) *Koch<NN>Mütze<+NN>*

returned by Morphisto, our algorithm correctly chose the second analysis (b).

Our approach disambiguates analyses of two possible candidates with an average precision of $\approx 64\%$, analyses of three candidates are disambiguated by an average precision of $\approx 48{,}8\%$ and for those in four ways ambiguous, there was an average precision of $\approx 55\%$.

We noticed two main types of disambiguation errors. The first problem is data sparseness, which can be handled by smoothing. The second problem is due to the fact that our model does not take into account the structuring of constituents. We are thinking about extending our PoS-based model, so that the semantic relatedness between Head-Modifier pairs can be expressed. This can be achieved by using semantic classes like WordNet.

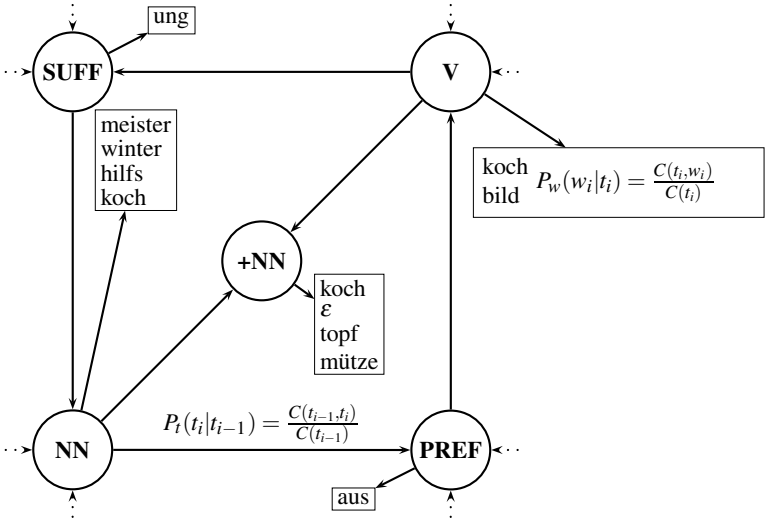

Fig. 1. Hidden Markov Model for disambiguation with transition probability P_t and lexical probability P_w

5 Applications

5.1 Integration into the TextGrid Workbench

TextGrid offers an Eclipse-based rich-client platform as development environment for all text-centered scientific disciplines. Collaborative work is supported by the grid, where resources (storage and computing power) can be shared easily among scientists over the net. Additional functionality is provided through a series of web services, among them Morphisto, that can be easily used from within the working environment. In fact, the German analyzer Morphisto marks a cornerstone in TextGrid, as it addresses linguists, philologists, and lexicographers alike. Moreover, the Morphisto web services will be used in a number of different applications in TextGrid. A general overview on the TextGrid architecture is shown in Figure 2.

5.2 Lemmatization and Morphological Analysis

Morphisto can be used to analyse in batch mode where the input is either plain text, a tokenized word list, or XML-encoded text. The tool produces either a 'simple' annotation or a TEI-style annotation. Moreover, single words can be analysed interactively. For instance, when working on a file in an (XML-)editor, the user can double-click on a word, and then the Morphisto New High German lemmatizer is called upon or, alternatively, a look-up function on historic forms is performed.

5.3 Generating Wordforms for Elexiko

Within the German ELEXIKO project [11], a corpus-based lexicon project that aims to build up rich lexicographic resources, lexicographers try to induce morpho-syntactic

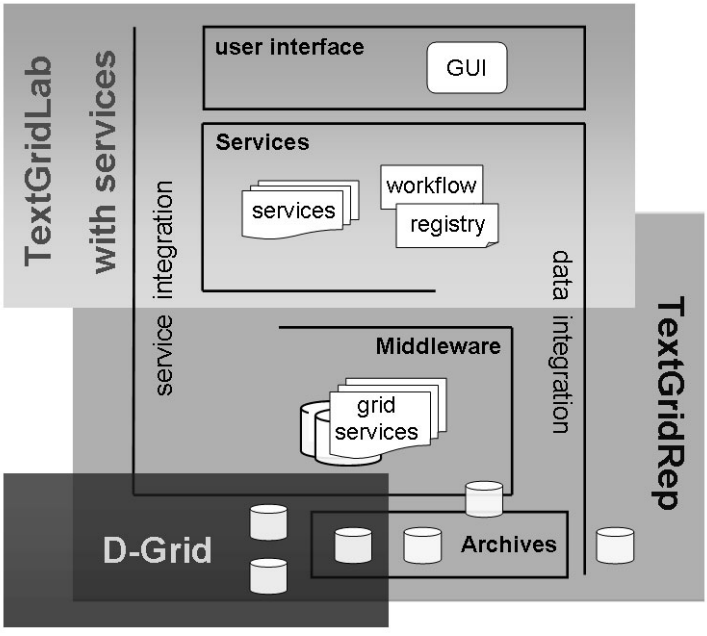

Fig. 2. TextGrid Architecture

information (semi)automatically by using Morphisto. For instance, the elexiko lemma list covering 300,000 word entries has to be enriched with the corresponding inflectional paradigm. To this end, the paradigm of a complex lemma is, as far as possible, derived from the corresponding head element. Finally, each entry will be reconfirmed by the lexicographer and it will be then stored in the database.

5.4 Deriving the Structure of Complex Entries

Another application on top of Morphisto is the automatic generation of hierarchically structured morphological trees. The algorithm delivers suggestions for the right/left-branching structure of complex words, which are to be confirmed by the lexicographer and added into the elexiko database. To this end, a user-interface for correcting false analyses has been built.

Approaches to semi-automatic detection of internal word structures have been followed in [12]. Here, a machine-learning approach with feature vectors for each right and left context indicating possible segmentation boundaries is proposed. Marek [13] uses a weighted finite-state approach with probabilities trained on a decomposed and bracketed list of compounds from CELEX. The first system achieves an accuracy of approx. 87% correctly segmented words, while Marek's system reports on a recognition rate of 96%.

In contrast to the systems mentioned above, our task is simpler. It relies on the given Morphisto analysis, such that problems related to linking elements/ alternations between constituents as well as inflections can be ignored.

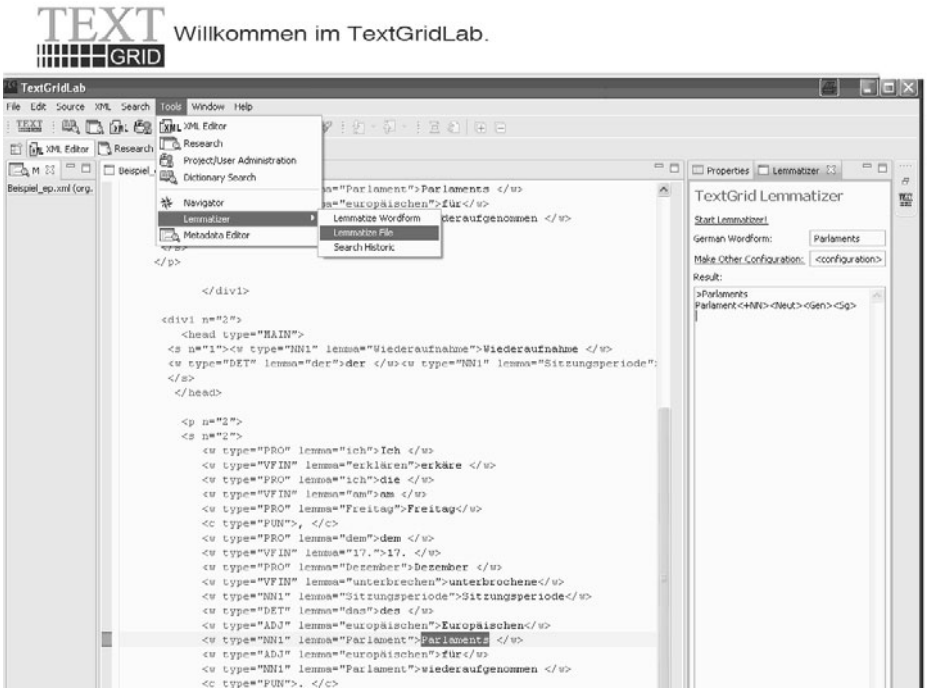

Fig. 3. TextGrid XML-Editor with Integrated Lemmatizer Function

Our method works bottom-up and computes the tree structure in two main steps. In the first step, the Morphisto analysis is split up and the routine tries to build the tree structure by attaching all suffixes and prefixes recursively from the inside out. The second step starts when there are no more nodes to be added. To complete the tree structure, the routine looks at which constituents can be put together, by looking up the word in the elexiko lemma-list. If there is more than one possible way for creating a new node, the constituent that occurs more often in the list is chosen. This step is repeated until there are just two nodes left. These last two nodes are simply connected to the root node.

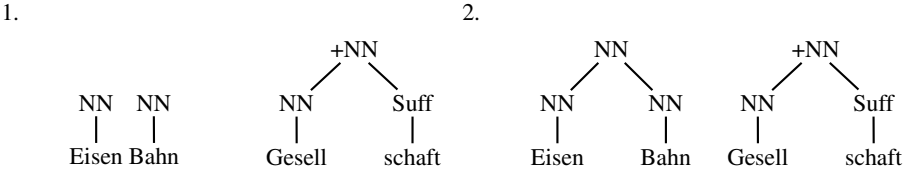

Fig. 4. First and second step of the routine (*Eisen-Bahn* occurs more often than *Bahn-Gesellschaft* (zero times))

In [14] the test results are presented in more detail. We want to show that routine in an example in Figure 4.

Eisen<NN>Bahn<NN>Gesell<NN>schaft<SUFF><+NN>

6 Technical Aspects: Morphological Analysis in the Grid

From the very beginning, Morphisto was meant to be an integral part of the philological toolchain provided by TextGrid for everyday use in eHumanities tasks. However, in compliance with the grid philosophy, it was not supposed to be shipped with the philological workbench *TextGridLab* as a piece of software that runs locally on the workstation consuming precious memory space. Instead, it was meant to be a web service running on a geographically distant grid server, which is accessible from within the philological working environment through a graphical user interface. And this procedure has payed off: All SFST compact format based automatons used by Morphisto in conjunction with TextGrid have a total size of ca. 150 MB (24 MB alone for the standard DeReWo-based MORPHISTO automaton as distributed open-source over the Internet). Furthermore, these automatons can be read and processed by the SFST tools only (mainly fst-infl2, fst-mor), which would have required platform-dependent binaries of the SFST tools for every platform, plus a Java binding. But by implementing a document-style web service complying with the WSDL 1.1[6] standard using Python 2.4 in conjunction with ZSI 2.0 and SOAPpy[7], we could provide easy, platform-independent access to Morphisto on a high-performance Linux machine to all clients capable of WSDL web services. That way, we could easily integrate Morphisto into the Eclipse rich-client application, *TextGridLab*, by implementing an Eclipse plugin that uses the Java version of Axis2[8] to access the Morphisto Web Service.

But on the server side we were faced with obstacles grounded especially in performance and concurrency. We had to make it possible for the web services to take multiple requests for single-word or batch lemmatization/morphological analysis and at the same time providing a quick response. Depending on the input type and the desired configuration, we follow two different strategies: calling the *fst-infl2* command from the SFST toolkit or calling the lemmatizer daemon.

The first strategy mainly applies to large, pre-tokenized input files. The SFST command *fst-infl2* reads a specified automaton in compact format, and an input file with German word tokens runs the tokens through the automaton and produces an output file with the results. Although this strategy requires a certain amount of time for reading the automaton and post-processing the output, it is still the fastest way for input files with at least 1,000 word tokens without any special configuration options. There are certain configuration options (e.g., “Use Guesser for Unknown Wordforms” or “Use Fuzzy

[6] W3C: Web Services Description Language (WSDL) 1.1.W3C Note 15 March 2001, available from `http://www.w3.org/TR/wsdl`

[7] Zolera SOAP Infrastructure (ZSI) and SOAPpy are available from the Python Web Services Project at `http://pywebsvcs.sourceforge.net`

[8] Axis2/Java is available from the The Apache Software Foundation at `http://ws.apache.org/axis2`

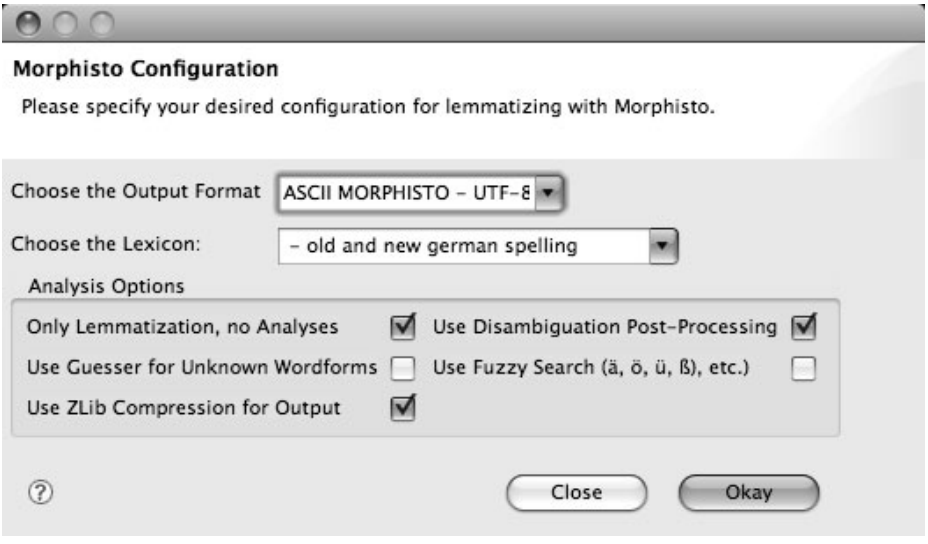

Fig. 5. Configuration option of Morphisto in TextGrid

Search", cf. Figure 5 for available configuration options of the Morphisto web service) that require the web service to interact with the analysis process: If the first processing of the wordform yields no result, the input will be routed through a different automaton. That procedure is not possible with *fst-infl2* in its original form. However, we managed to patch *fst-infl2*, such that it is possible for the Python web-service to "chat" with *fst-infl2* via the standard input and standard output on UNIX machines. This method has been useful for processing TEI input files and annotating them with Morphisto output.

For small input files, it is not efficient to load an automaton into memory and process merely one single word form. For theses situations, we have devised a daemon that communicates via a UNIX socket port on the server with the web service. The advantage of this second strategy is that the time-consuming task of reading automatons from a file is done only once – at startup of the daemon – and can be avoided at each request. However, this method has proven to be not all efficient for processing large input files. This may be mainly due to two reasons: Firstly, the inter-process communication between daemon and web-services uses the TCP-protocol[9] for flow control. In contrast to UDP[10], which was tested in prior experiments, it has proven to work reliably. Secondly, the daemon is written in Python and depends on Toni Arnold's Python binding for the C++ based SFST toolkit[11] for reading automatons and applying the queried wordforms

[9] RFC 793: Transmission Control Protocol (TCP), available from `http://tools.ietf.org/html/rfc793`

[10] RFC 768: User Datagram Protocol (UDP), available from `http://tools.ietf.org/html/rfc768`

[11] Toni Arnold's Python binding for SFST is available from `https://gna.org/projects/pysfst`

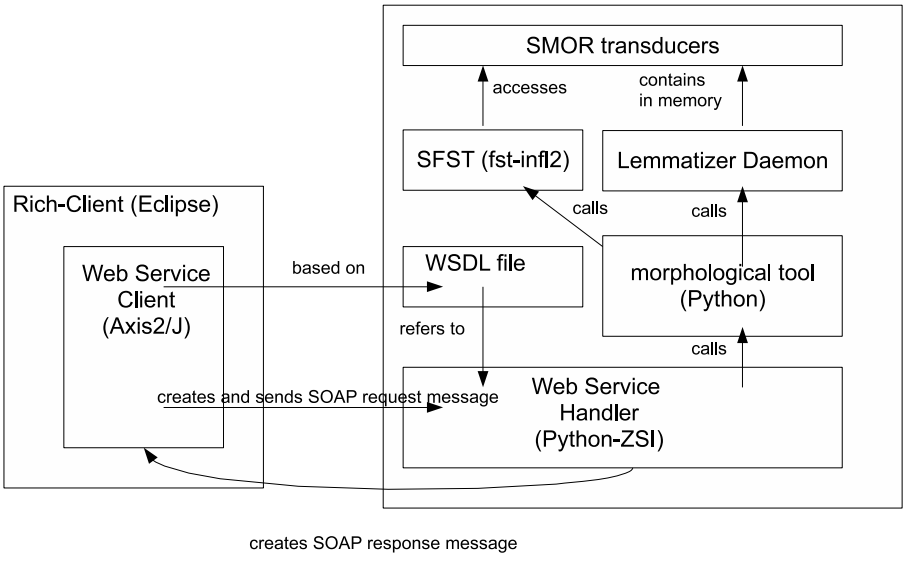

Fig. 6. Integration of Morphisto into TextGrid

on them. Maybe, the effort of developing a C++-based solution in the first place would have payed off in the long run in terms of performance and elegance. And yet, also in this case Python allowed us to come up with a solidly working solution in very little development time. Especially the asyncronous chat module has solved our concurrency issues we used to experience earlier[12].

For a schematic of how Morphisto is integrated in TextGrid, please see Figure 6.

7 Summary and Outlook

This paper presented the design of Morphisto as an integrated web service within the eHumanities platform TextGrid as well as recent developments in terms of specific applications of Morphisto. The development of Morphisto continues as an open-source project hosted at *Google Code*[13]. Everybody with a non-commercial interest in Morphisto is welcome to participate and contribute. At present, the ELEXIKO project makes the effort of manually enhancing the Morphisto lexicon and slowly building up a collection of gold standard analyses for complex German wordforms. The integration of the disambiguation function into Morphisto by use of a weighted finite state transducer is currently under development. Moreover, we plan to use semantic criteria for disambiguation based on WordNet classes.

[12] Documentation for *asynchat* is available in the Python Standard Library Reference at `http://www.python.org/doc/2.5.4/lib/lib.html`

[13] `http://code.google.com/p/morphisto`

References

1. Gietz, T., et al.: TextGrid and eHumanities. In: Sloot, P. (ed.) IEEE Conference on e-Science and Grid Computing, Amsterdam, December 4-6, p. 133. IEEE Computer Soceity, Los Alamitos (2006)
2. Yli-Jyrä, A.: Toward a Widely Usable Finite-State Morphology Workbench for Less Studied Languages – Part I: Desiderata. Nordic Journal of African Studies 14(4), 479–491
3. Schmid, H.: A Programming Language for Finite State Transducers. In: Yli-Jyrä, A., Karttunen, L., Karhumäki, J. (eds.) FSMNLP 2005. LNCS (LNAI), vol. 4002, pp. 308–309. Springer, Heidelberg (2006)
4. Schiller, A.: Deutsche Flexions- und Kompositionsmorphologie mit PC-KIMMO. In: Hausser, R. (ed.) Linguistische Verifikation. Dokumentation zur ersten Morpholympics 1994 (1996)
5. Heid, U., Säuberlich, B., Fitschen, A.: Using Descriptive Generalisations in the Acquisition of Lexical Data for Word Formation. In: Proceedings of the 3rd Conference on Language Resources and Evaluation, Las Palmas de Gran Canaria, Spain, vol. IV, pp. 86–92 (2002)
6. Zielinski, A., Simon, C.: Morphisto: An Open-Source Morphological Analyzer for German. In: Proceedings of the FSMNLP 2008, pp. 177–184 (2008)
7. Jacke, B.: Deutsches Wörterbuch für Ispell, Myspell und Hunspell nach den neuen Rechtschreibregeln (2006), `http://www.j3e.de/ispell/igerman98/`
8. Schmid, H., Fitschen, A., Heid, U.: SMOR: A German Computational Morphology Covering Derivation, Composition, and Inflection. In: Proceedings of the IVth International Conference on Language Resources and Evaluation (LREC 2004), Lisbon, Portugal (2004)
9. Schmid, H.: Disambiguation of Morphological Structure Using a PCFG. In: Proceedings of the Human Language Technology Conference and the Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP 2005), Vancouver, Canada (2005)
10. Geyken, A., Hanneforth, T.: TAGH: A Complete Morphology for German based on Weighted Finite State Automata. In: Yli-Jyrä, A., Karttunen, L., Karhumäki, J. (eds.) FSMNLP 2005. LNCS (LNAI), vol. 4002, pp. 55–66. Springer, Heidelberg (2006)
11. Klosa, A., Schnörch, U., Storjohann, P.: ELEXIKO: Lexical and Lexicological, Corpus-based Hypertext Information System at the Institut für deutsche Sprache, Mannheim. In: Proceedings of the 12th Euralex International Congress (2006)
12. Daelemans, W., van den Bosch, A.: Memory-Based Language Processing. Cambridge University Press, Cambridge (2005)
13. Marek, T.: Analysis of German Compounds Using Weighted Finite State Transducers, Bachelor thesis, University of Tübingen (2006)
14. Wittl, T.: German Morphological Disambiguator based on statistical data. In: Proceedings of TaCoS 2009 (forthcoming, 2009)

Morphosyntactic and Semantic Analysis of Text: The MPRO Tagging Procedure

Heinz Dieter Maas, Christoph Rösener, and Axel Theofilidis

IAI (Institute for Applied Information Science)
Martin-Luther-Str. 14
D-66123 Saarbrücken, Germany
{dieter,chrisr,axel}@iai.uni-sb.de

Abstract. The MPRO system is a package of programs developed by the Institute for Applied Information Science (IAI) to perform morphological, syntactic and semantic analysis for several languages. Especially for German the MPRO system is a very efficient and robust language software tool. The linguistic analysis within the system is split up into several sub-tasks. In this paper we will present the general MPRO tagging procedure and its sub-tasks in some detail. We will take a closer look at the single program modules and their particular output. The function of each module is described in detail together with the possible parameters of the features given in the particular output. Furthermore the description of each module is enriched with manifold practical examples to demonstrate the functionality of the system. Finally some possible areas of application of the MPRO software package are stated.

1 Overview

MPRO is a program package developed by the IAI [7] which is able to perform morphological analysis for a dozen languages, mainly German, English, French and Spanish [8]. The analysis results can be the basis for parsing, spell checking and grammar checking, tagging, indexing, abstracting and other text mining routines.

As a major improvement of MPRO during the last four years, a large-coverage lemma dictionary has been set up that is looked up during morphological analysis complementary to the morpheme dictionary. On this basis, accuracy of morphological analysis has been improved in different respects, as we will show.

The MPRO process is split up into the following sub-tasks:

- Module LESEN: Reading and normalization of the input text, identification of sentence boundaries and recognition of character strings which are presumed to be word forms. These strings are submitted to different analysis modules including morphological analysis. LESEN needs a lexical data-base.
- Module LEXNP: Addition of information with respect to possible fixed phrases (NPs). LEXNP uses a special dictionary (available only for German).
- KOMPLETT: This program treats a very special and rather frequent German phenomenon, namely the deletion of a head or a non-head in a coordination of compound nouns. KOMPLETT tries to reconstruct the deleted element.

C. Mahlow and M. Piotrowski (Eds.): SFCM 2009, CCIS 41, pp. 76–87, 2009.

- KORRIGIERE: If LESEN cannot analyse a character string, it will be marked as an unknown word. KORRIGIERE determines whether such a string is possibly a misspelled word and tries to propose a spelling correction. The program needs a list of word forms and the lexical data-base used in LESEN.

All these modules are programmed in C and can run under UNIX and Windows. We will present below the single modules and their output in some detail.

2 The MPRO Tagging Procedure

2.1 The Module LESEN

LESEN expects 'normal' text, i.e., an ASCII file in plain text or HTML format. When treating an HTML text, LESEN uses a file referred to as "sgmlmarks" which lists the so-called "internal tags", e.g., <i>, <b> or <u>. These tags relate to words in some way and do not interrupt a sentence, whereas unknown tags are considered as "external" and mark the beginning or the end of a sentence. Special tags listed in "sgmlmarks" may be annotated with specific instructions regarding the processing of the tag content. For instance, the instruction `oneword:` can be used to treat the tag content as a single word unit. In addition, LESEN knows about all kinds of entities for the transliteration of special characters (like ü for ü or   for a non-breaking space) in texts of both HTML and plain text format.

After the text is "purified", i.e., tags are given an internal representation and special characters are transliterated, we use blanks and tabs for the segmentation of character strings. If a string contains punctuation marks, the segmentation continues. All resulting segments and combinations thereof might be "words". Which particular combinations should be tested, is determined on the basis of pattern descriptions contained in a file called "structurerules". This file also comprises patterns for the recognition and handling of special objects like e-mail addresses or dates.

For further analysis, a function ("lookup_lex") is called which takes as input a string (and a pointer to all existing lexical resources for the language treated) and returns either "unknown" or a set of structures which describe the linguistic properties of the string. This function is used at different places in LESEN. For instance, we may be forced at some point in the analysis procedure to split an unknown word and to analyse the pieces in order to find out that there is a blank missing between two words, for instance, when a digit and a unit are written with no blank as in *50mph*.

The function lookup_lex fulfils the following subtasks:

- Find a word in the cache storage (used for efficient performance). If found, break.
- Find a word in the word form dictionary morph.fph (covering closed class elements, e.g., articles or prepositions, as well as fixed phrases)
- Find a word in the user dictionary user.fph
- Identify a word by means of morphological analysis
- Put the result into the cache

With regard to the special dictionary morph.fph there exists a function which identifies sequences of words as (non-inflecting) fixed expressions. Such sequences are considered as one word, e.g., *all day long* and *for example* in English, *afin que* and *au bout de* in French, or *ab sofort* and *ab und zu* in German.

The information resulting from the analysis steps sketched above is represented as a set of feature bundles (FBs), where an FB is a sequence of attribute-value pairs. If the set contains more than one FB, the word is ambiguous as is illustrated by the following set of partial FBs for English *sample*:

```
{ori=sample,snr=1,wnra=1,wnrr=1,pctr=no,pctl=no,last=ye
s,gra=small,c=verb,vtyp=inf,lu=sample,w=1,cs=v,ds=sampl
e,ls=sample,s=nil,ts=sample,t=sample}
{ori=sample,snr=1,wnra=1,wnrr=1,pctr=no,pctl=no,last=ye
s,gra=small,c=verb,vtyp=fiv,tns=pres,lu=sample,w=1,cs=v
,ds=sample,ls=sample,s=nil,ts=sample,t=sample}
{ori=sample,snr=1,wnra=1,wnrr=1,pctr=no,pctl=no,last=ye
s,gra=small,c=noun,nb=sg,case=nom;acc,s=act,lu=sample,w
=1,cs=n,ds=sample,ls=sample,ts=sample,t=sample,ehead={c
ase=nom;acc,nb=sg}}
```

The following features which reflect textual, contextual or graphical properties are always present in an FB:

- **ori:** The word form of the text, as it was submitted to lookup_lex or a concatenation of word forms if this entity was recognized as a fixed phrase.
- **snr:** Sentence number.
- **wnra:** Absolute word number (numbering throughout the text).
- **wnrr:** Relative word number (numbering starting anew for each sentence).
- **pctr:** Value 'yes', if the next unit is a punctuation mark.
- **pctl:** Value 'yes', if the preceding unit is a punctuation mark.
- **last:** Marks the last word of a sentence, if its value is 'yes' (otherwise 'no').
- **gra:** Shows the graphics of the word, e.g., only lower case letters; one upper case letter followed by lower case letters; only upper case letters; only digits; letters and digits mixed; other characters.
- **hyphen:** Indicates whether the word contains a hyphen.

Other features are assigned on the basis of lexicon lookup and morphological analysis.

3 Morphological Analysis

"Morphological analysis" is a program within LESEN which contains general morphological rules for the different languages in the form of subroutines. All kinds of word formation patterns (compounding, derivation, prefixation, inflection) are covered. A large number of operations are language independent, but there also exist rules explicitly designed for specific languages.

3.1 The Lexical Database lexdb

Morphological analysis co-operates with different types of lexica which are maintained in a larger lexical database called **lexdb**. On the one hand, this lexical database includes minor types of entries such as those for inflectional and derivational endings or prefixes. The majority of entries, however, is of one of the following two major types:

- **simple lexeme** encoding the morphological characteristics of one or more stem morphemes; among the entries of this type are also a large quantity of toponyms and other names
- **complex lexeme** representing a word resulting from word formation processes such as compounding, derivation, or prefixation, and being characterized by its internal morphological structure as is delivered by morphological analysis based on LESEN

The German lexdb currently contains more than 736,000 entries, among which are almost 120,000 entries representing simple lexemes, i.e., morphological entries. All entries are represented as feature bundles as is illustrated by the following (partially depicted) entries for the German nouns *Apfel* and *Baum* 'apple' and 'tree'.

```
{string=apfel,c=n,g=m,lng=germ,s=thing&ed,m={stem=apfel
,f=fs-no,a={ig=na},fuge=0};{stem=äpfel,f=no-f0n,n={chen
=dim},fuge=0},mstru={g=m,ds=apfel,t=apfel,ss=thing&ed},
wf={form=Apfel;Apfels;Äpfel;Äpfeln}}
{string=baum,c=n,g=m,lng=germ,s=tree,m={stem=baum,f=fs_
es_e-no,fuge=0};{stem=bäum,f=no-fe,n={chen=dim,ge0=set}
,fuge=e},mstru={g=m,ds=baum,t=baum,ss=tree},wf={form=Ba
um;Baumes;Baums;Bäume;Bäumen}}
```

As opposed to the subsequently shown entries, the above entries are of type **simple lexeme**. As their main characteristic, they comprise the feature `m={...}` which encodes the morphological properties of the two allomorphic stems *apfel* and *äpfel* resp. *baum* and *bäum* (cf. below for more details).

The following entries for the German compound *Apfelbaum* 'apple tree' and the derived (diminutive) noun *Äpfelchen* 'small apple', on the other hand, are typical examples of entries of type **complex lexeme**. Representing a compound respectively a derivation, they do not comprise the feature `m={...}`, whereas the feature `mstru={...}` reveals their complex morphological structure:

```
{string=apfelbaum,c=n,g=m,lng=germ,mstru={g=m,gs=m#m,ds
=apfel#baum,t=apfel#baum,ss=thing&ed#tree},wf={form=Apf
elbäume;Apfelbäumen;Apfelbaum;Apfelbaums}}
{string=äpfelchen,c=n,g=n,lng=germ,mstru={g=n,ds=apfel~
chen,t=äpfelchen,ss=thing&ed},wf={form=Äpfelchen}}
```

The lexical database **lexdb** is elaborated and maintained by lexicographers, but it is not directly accessed during the morphological analysis procedure. For runtime use, it is instead transformed into two distinct dictionaries: the morphological dictionary **morpho** and the lemma dictionary **lemwb**.

3.2 The Dictionaries morpho and lemwb

The dictionary **morpho** is a transformation of the set of simple lexemes. Only morphological information is transferred into morpho. Thus, for instance the entry for *Apfel* shown above will be transformed into the following two entries to be used at runtime:

```
{string=apfel,lu=apfel,c=n,g=m,lng=germ,s=thing&ed,f=fs
–no,a={ig=na},fuge=0}
{string=äpfel,lu=apfel,c=n,g=m,lng=germ,s=thing&ed,f=no
–f0n,n={chen=dim},fuge=0}
```

Each such entry describes how the respective stem (now shown in the feature `string`) behaves morphologically, i.e., we find information about inflectional paradigms (e.g., `f=fs–no` or `f=no–f0n`), possible derivations (e.g., `a={ig=na}` producing the adjective *apfelig* or `n={chen=dim}` producing the diminutive noun *Äpfelchen*) and binding elements (e.g., `fuge=0` used for compounding as in *Apfelbaum*). Beyond these basic types of features you also find indications on the use of prefixes, morphological negation elements etc.

To reduce overgeneration we can also prohibit prefixes or certain nonsensical compounds by means of special features. Moreover, many entries have etymological and semantic information which, however, is only partly used in analysis.

The dictionary **lemwb,** on the other hand, is a transformation of both simple and complex lexemes. All information valid at lexeme (not morpheme) level is transferred into lemwb entries.

```
{string=apfel,c=n,g=m,lng=germ,s=thing&ed,mstru={g=m,ds
=apfel,t=apfel,ss=thing&ed}}
{string=apfelbaum,c=n,g=m,lng=germ,mstru={g=m,gs=m#m,ds
=apfel#baum,t=apfel#baum,ss=thing&ed#tree}}
```

Note that information concerning inflection or other morphological features need not be represented in **lemwb** entries since, during morphological analysis, this information is propagated from the respective head element, be it a stem as, for instance, *baum* in *Apfelbaum*, or a derivational ending as, for instance *–chen* in *Äpfelchen*.

The benefit obtained from the **lemwb** dictionary will become apparent when we consider the morphological analysis procedure in some more detail.

3.3 Morphological Analysis Procedure

An input string is analyzed by a sequence of operations:

- Segmentation of the input string. This results in a coherent graph with one start and one end node. The arcs are marked by strings which were found in the morphological dictionary. The segmentation procedure uses a list of "wrong" pairs of strings which helps to avoid segmentations that will never produce a result, e.g.,

```
...
abs|tand
abt|ei
abt|eil
...
```

- Lookup of the strings in the **morpho** dictionary, assignment of the information found.
- Application of the **analysis rules, e.g.**, for derivation, compounding, inflection.

Analysis rules are functions with one, two or three arguments. They check the entry conditions and produce one intermediate result or nothing. As the input is normally ambiguous, all rules are tentatively applied to all sub-paths of the graph with one, two or three elements. A positive result is added to the graph, and therefore the new object will be treated as well in further processing.

For instance, by application of a general compounding rule taking two stem morphemes as arguments we obtain the following intermediate result from analysing the German compound noun form "Apfelbäume":

```
...
<0>Apfelbäum–n<9>:{c=n,g=m,lng=germ,s=tree,lu=apfelbaum
,f=no-fe,fuge=e,w=2,cs=n#n,gs=m#m,t=
apfel#baum,ts=apfel#baum,ds=apfel#baum,ls=
apfel#baum,lngs=germ#germ,lexfound=yes,zf=
Apfelbaum,ss=thing&ed#tree,lexical={lu=apfelbaum,c=n}}
...
```

Such intermediate results of morphological analysis are not immediately inserted into the graph, but are first looked up in the **lemwb** dictionary. If you consider the feature bundle for the partial analysis of "Apfelbäume", you will see the feature 'lexfound=yes' as a trace of this intermediate result being successfully looked up in **lemwb**.

Successful lookup in the **lemwb** dictionary may have several effects contributing to an increased accuracy of morphological analysis:

- Morphological features that are propagated by default in analysis rules can be overwritten by encoding additional information in **lemwb** entries in the feature `m2={}`. For instance, restrictions on compounding elements (feature `fuge`) can be relaxed at compound level. Whereas the noun *Hund* requires *–e* or *–s* as compounding element, the compound *Schäferhund* also allows the null-element:

  ```
  {string=schäferhund,c=n,g=m,lng=germ,m2={fuge=0;e;s},ms
  tru={g=m,gs=m#m,ds=schaf~er#hund,t=schäfer#hund,ss=agen
  t&isto#animal}}
  ```

- Furthermore, a whole range of second order morphological phenomena can only be effectively handled at the level of **lemwb** entries. To give just two examples:

  Unless the adjective *zurechnungsfähig* is lexicalized, the negated form *unzurechnungsfähig* can only be handled at the level of compounding, since there is no word like *Unzurechnung*; thus the **lemwb** entry with the respective `m2`-feature:

  ```
  {string=zurechnungsfähig,c=a,lng=germ,m2={neg=un},mstru
  ={gs=f#u,ds=zu_$rechnen~ung#fähig,t=zurechnung#fähig,ss
  =ation#a}}
  ```

Unless the verb *menschlichen* is lexicalized, the prefixations *entmenschlichen* and *vermenschlichen* can only be handled at the level of derivation; thus the **lemwb** entry for the adjective *menschlich* with the respective m2-feature:

```
{string=menschlich,c=a,lng=germ,m2={v={ent_0=remove,ver
_0=turn},mstru={ds=mensch~lich,t=menschlich,ss=agent}
```

- Finally, entries in the **lemwb** dictionary may be marked as nonsensical (nonsense=yes) such that respective analysis results will be deleted when matching. For instance, the analysis corresponding to the following entry which represents a nonsensical compound analysis of *Emailleurin* will be discarded to the benefit of obtaining only the analysis corresponding to the derivational analysis of *Emailleurin* shown next:

```
{string=emailleurin,c=n,g=m,lng=germ,syn={s=mat},mstru=
{g=m,gs=f#m,ds=emaille#urin,t=emaille#urin,ss=mat#mat},
nonsense=yes}
{string=emailleurin,c=n,g=f,lng=fr,syn={s=agent&isto},m
stru={g=f,ds=emaille~eur~in,t=emailleurin,ss=agent&isto
}}
```

When the rule system can no longer produce results, the graph is checked for arcs starting with the in-node and ending with the out-node. All these arcs contain the final result(s) for the string analyzed. If there is no such result, the morphological analysis of the input string failed.

As mentioned, the results of morphological analysis are represented as a set of feature bundles, as shown above. In what follows we list a few features produced by morphological analysis:

Features for Nouns

- Attribute **c:** Category. In this case the value is **noun**.
- Attribute **g:** Gender. Values are **m** (masculine), **f** (feminine) and **n** (neuter).
- Attribute **nb:** Number. Values are **sg** and **plu**.
- Attribute **case:** Case. Values are **nom**, **gen**, **dat** and **acc**.

If values are ambiguous, they may be combined as a disjunction. In fact, the values depend partly on the language, e.g., Russian has more cases, French fewer genders, etc.

Features for Adjectives

- Attribute **c:** Category. In this case the value is **adj**.
- Attribute **deg:** Degree. Values are **base**, **comp** and **sup**.

Depending on the language, case, number, gender, and inflectional type (strong vs. weak inflection, for German) will be encoded in respective features. German adjectives with null-ending are considered adverbs (they can function as adverbs or as predicatives).

Features for Verbs

- Attribute **c**: Category. In this case the value is **verb**.
- Attribute **vtyp**: Subtype. Values are **fiv** (finite), **inf** (infinitive), **ptc1** (present participle), **ptc2** (past participle), **izu** (infinitive with infixed *zu,* for German), **imperative** , **ger** (gerund) and others, depending on the language.
- Attribute **tns**: Tense. Values may be **pres**, **past**, **imp**, **pret**, **fut**, depending on the language.
- Attribute **nb**: Number (as for nouns).
- Attribute **per**: Person. Values are **1**, **2**, and **3**.
- Attribute **mode**: Mood. Values are **ind** (indicative), **subj** (subjunctive), etc.

Category-Independent Features. There exist some category-independent features which we introduce here with reference to the partially depicted feature structure obtained by analysis of the word *Polyvinylchloride*:

```
{ori=Polyvinylchloride,lu=polyvinylchlorid,c=noun,nb=pl
u,case=nom;gen;acc,g=n,s=mat,lng=neolat,w=3,ts=poly#vin
yl#chloride,t=poly#vinyl#chlorid,cs=a#n#n,gs=u#n#n,ds=p
oly#vinyl#chlor~id,ls=poly#vinyl#chlor,lngs=lat#neolat#
gr,ss=a#mat#mat}
```

- The attribute **lu** shows the citation form of the word (*lexical unit*); it serves as a lookup key to the **lemwb** dictionary
- The attribute **s** describes the semantics of the compound word as a whole. The value **mat** means, that *Polyvinylchlorid* is a (non-edible) material.
- The attribute **lng** gives etymological information.
- The attribute **w** describes how many elements constitute a word (in our case 3). If w > 1, the word is a compound.

A group of features describes the word structure and its semantics. Parts characterizing the elements of a compound are separated by # in the feature value:

- The attributes **ts** and **t** show the surface segmentation of the word and the corresponding lexical unit elements.
- The attribute **cs** indicates the categories per element of a word; in our example, **a** for adjective and **n** for noun.
- The attribute **gs** indicates the gender per element of a word; in our example, **u** for unspecified and **n** for neuter.
- The attribute **ds** indicates the derivational structure per element of a word; in our example, the element *chlorid* is derived from the root form *chlor*; derivational elements are attached by ~.
- The attribute **ls** indicates the root words of the elements of a word
- The attribute **lngs** indicates the etymology of the elements of a word
- The attribute **ss** indicates the semantics of the elements of a word

If desired, morphological analysis can return a feature called **lexical** which keeps track of partial compounds that have matched during lookup in the **lemwb** dictionary. On this basis it can be revealed that *Polyvinylchloride* is, in fact, ambiguous.

- 1st result: lexical={lu=polyvinylchlorid,c=n};{lu=vinylchlorid,c=n}
- 2nd result: lexical={lu=polyvinylchlorid,c=n};{lu=polyvinyl,c=n}

3.4 The Module LEXNP

LEXNP is totally language-independent. It uses only information found in a special dictionary representing multi-word structures. The program tries to find a piece of text matching a dictionary entry which describes, e.g., a noun phrase. As a result, all the FBs matched receive additional information. Existing features cannot be overwritten.

This procedure can be used for preparing the basis of checking capitalization within multi-word units, e.g., *das Rote Meer* 'Red Sea' is correct in German, although *Rote* is an adjective and should be written with a lower case letter. Inversely we could indicate that in *das rote Meer* the word *rote* should be capitalized.

3.5 The Module KOMPLETT

KOMPLETT is used for the analysis of German texts only. In German we frequently find the phenomenon of deletion of pieces of compounds, if they are co-ordinated. Graphically the missing element is represented by a hyphen. Examples:

- *Firmen- und Privatkapital* (Firmenkapital)
- *Unternehmensplanung und -steuerung* (Unternehmenssteuerung)
- *raum- und zeitbezogene, raum- und zeitbezogen* (raumbezogen)
- *Vor- und Nachlaufachse* (Vorlaufachse)
- *auf- und zugeklappte* (aufgeklappte)

If a word ends with a hyphen, the program checks whether the next word is a coordinator and examines the following word. If it starts with a hyphen, on the other hand, the text to its left hand side is checked. If the incomplete word in question can be completed, the result is stored in its FB and the two co-ordinated words receive a feature showing that they form a co-ordinated pair. The incomplete word receives a new `lu`-feature, e.g., `lu=firmen-` is changed to `lu=firmenkapital`. The word structure (the features **t**, **ts**, **ds** etc.) is also changed accordingly. If a word cannot be completed, this may lead to an error message in a grammar checker.

The conditions under which the completion process is performed is in fact much too restrictive. If the text is *Firmen- und auch Privatkapital*, we will not get a result for *Firmen-*. Moreover, the module does not use the lexical database and no morphological analysis checks the results. Therefore it may happen that the reconstructed word is wrong.

3.6 The Module KORRIGIERE

KORRIGIERE is designed for computing proposals for wrongly spelled words. It uses a table called **wordvar**, a huge list of correct words (called **wfkorr**, more than one million entries) and the whole morphology procedure with the dictionaries derived from **lexdb**.

KORRIGIERE starts to work if it finds a word with the feature `state=unknown`. However, if such a word is preceded by a word marked as a title or a first name, it is not treated, because it is probably a family name. If a piece of text between hyphens or citation marks contains unknown words, a check is done to determine whether this might be a citation in a foreign language. If this seems to be true, all the words of this unit will be marked by the feature `foreign=yes`, and consequently the procedure will not attempt to correct such words.

Below we show by means of an example, how the correction process works. The word in question is the misspelled *Kernpyhsik*.

- Check if the word can be found in the cache storage for names or already treated words. If so, break.
- We use the table **wordvar** for producing variants of the word:
  Kernpyhsik (i.e., the original)
  Kermpyhsik
  Kernpühsik
  Kernpyhsirk
  Kempyhsik
  Kernpyhsyk
  Kernpyhsiek
- All these words (except the first one, the original) are looked up in the word list **wfkorr**. Sometimes one of them is found, a proposal is therefore identified and the correction process can stop. In our example, none is found in **wfkorr**.
- The word variants are submitted to morphological analysis. As we get no result (all variants are wrong) we must continue.
- Now we try to analyze the variants by loosening the restrictions concerning the compounding elements encoded by the feature `fuge`. Again, no result is obtained for our example.
- Now we try to find a similar word (for all variants) in **wfkorr**, looking at the first letters of each word. We find *Kernphysik* with a similarity of 100 with respect to the original, and *Kernphysiker* and *Kernphysiken* with a similarity of 76. The comparison process uses a similarity function which returns values with a maximum of 100.
- We also try to find a similar word by inverting the variants (i.e., the original *Kernpyhsik* becomes *kishypnreK*). The results are *Kernphysik* (similarity=100) and *Kryophysik* (similarity=63).
- Results with too low similarities are discarded. Especially for short words we may get several results. In our example the only good result is *Kernphysik*.
- The good results are re-analyzed, because they have to conform to the orthography rules (e.g., Germany/Austria/Switzerland, progressive vs. conservative spelling or for English British/American). In our example nothing needs to be changed.
- At the end the program modifies the original FB. `state=unknown` is replaced by `state=proposal` (or `state=ff` for a word with a wrong compounding element) and the attribute `ns` contains all proposals ordered according to their similarity. The results are put into a cache storage.

The final result is:

```
{ori=Kernpyhsik,c=noun,lu=Kernpyhsik,s=n,ds=kernpyhsik,
ls=kernpyhsik,w=1,ehead={case=nom;gen;dat;acc,nb=sg;plu
,g=m;f;n,infl=weak;strong;null},lng=germ,wnra=1,snr=1,g
ra=cap,pctr=no,pctl=no,last=yes,wnrr=1,state=proposal,n
s=Kernphysik}
```

An example where the original unknown word is recognized by loosening the restrictions for compounding elements is *Terminsänderung* (*s* is wrong). This produces:

```
{ori=Terminsänderung,c=noun,lu=Terminsänderung,s=n,ds=t
erminsänderung,ls=terminsänderung,w=1,ehead={case=nom;g
en;dat;acc,nb=sg;plu,g=m;f;n,infl=weak;strong;null},lng
=germ,wnra=1,snr=1,gra=cap,pctr=no,pctl=no,last=yes,wnr
r=1,state=ff,ns=Terminänderung}
```

4 Conclusion

The MPRO software package performs a deep morphological, syntactic and semantic analysis of given texts. Especially for the German language it provides a detailed output of features. Some modules of the system even treat very special German phenomena, e.g., the deletion of a head or a non-head in a coordination of compound nouns. Furthermore the system proposes spelling corrections for German, if the analysed character string is marked as an unknown word. By providing these detailed results there are many conceivable usage scenarios for the MPRO system. On the one hand the analysis results can be the basis for parsing, spell checking and grammar checking systems [4] and also for tools to check style [9], gender [1,2] or the usage of controlled languages (e.g., Simplified English). On the other hand the output of the system can be used not only for tagging but also for linguistically intelligent indexing and abstracting [6]. In addition to this, the software could even be used in language learning [5], machine translation [3] and to support linguistically intelligent information retrieval [10] and other text mining routines.

References

1. Carl, M., Garnier, S., Haller, J., Altmayer, A., Miemietz, B.: Gendercheck in German Texts. In: Proceedings of the International Conference on Language, Politeness and Gender, Nordica, Helsinki (2004)
2. Carl, M., Garnier, S., Haller, J., Altmayer, A., Miemietz, B.: Controlling Gender Equality with Shallow NLP Techniques. In: Proceedings of COLING 2004, vol. II, pp. 820–826 (2004)
3. Carl, M., Way, A. (eds.): Recent Advances in Example-Based Machine Translation. Kluwer, Dordrecht (2003)
4. Bibliographisches Institut & F. A. Brockhaus AG, Duden Verlag Mannheim, books and software (2009), http://www.duden.de/produkte/index.php?suchwort=Software

5. Garnier, S., Tall, Y., Fissaha, S., Haller, J.: Development of NLP tools for CALL based on Learner Corpora (German as a foreign Language). In: Archer, et al. (eds.) Proceedings of the 2003 Corpus Linguistics Conference, UCREL technical paper no. 16, pp. 246–252. UCREL, Lancaster (2003)
6. Haller, J., Schmidt, P.: AUTINDEX – Automatische Indexierung. Zeitschrift für Bibliothekswesen und Bibliographie: Sonderheft 89, 104–114 (2006)
7. Maas, H.-D.: MPRO – Ein System zur Analyse und Synthese deutscher Wörter. In: Hausser, R. (ed.) Linguistische Verifikation. Dokumentation zur Ersten Morpholympics 1994, pp. 141–166. Niemeyer, München (1996)
8. Maas, H.-D.: Multilinguale Textverarbeitung mit MPRO. In: Lobin, G., et al. (eds.) Europäische Kommunikationskybernetik heute und morgen. KoPäd, München (1998)
9. Reuther, U.: CLAT – Qualität von Anfang an! In: Terminologie von Anfang an. DTT-/tekom-Tagungsakte. tekom, Weimar (2006)
10. Rösener, C.: Die Stecknadel im Heuhaufen. Natürlichsprachlicher Zugang zu Volltextdatenbanken. Peter Lang, Frankfurt am Main (2005)

Word Manager

Pius ten Hacken

Swansea University, Department of Modern Languages,
Singleton Park, Swansea SA2 8PP, UK
p.ten-hacken@swansea.ac.uk

Abstract. Word Manager (WM) is a system for the specification and use of morphological knowledge. It is based on a problem analysis that departs from the usual distinction between a rule component and a lexicon. Instead, it integrates rules and lexical knowledge for morphology in an expert workbench from which components for the performance of certain operational tasks (e.g. analysis or lemmatization). The WM formalism and specification environment have been developed in such a way that the support for individual tasks is optimized. The rule formalism incorporates separate rule types for inflection, word formation, and spelling (i.e. where morphology is not purely concatenative) in the WM core. Phrase Manager includes rules for clitics, periphrastic inflection, and multi-word units. These rules make it possible to encode also German separable verbs adequately. Lexicographic specification is distinct from rule specification and exploits the rules to increase speed and consistency. WM components have been used in various environments.

1 Introduction

Word Manager (WM) is a system for morphological dictionaries. A morphological dictionary is a database in which lexical knowledge is organized according to morphological rules. In order to understand this concept fully, it is essential to see the underlying analysis of the problem of Natural Language Processing (NLP). Therefore section 2 is devoted to the problem analysis. WM has a number of different rule types, each adapted to the specific function they have. They are described in section 3. It is well known that the main source of poor quality in the specification of lexical databases is the lack of adequate support in coding. For this reason, the WM specification and maintenance environment has been developed in such a way that each type of user is optimally supported in their task. The implementation of this idea is described in section 4. Finally, section 5 gives an overview of currently available resources.

2 Problem Analysis

Traditionally, NLP systems are divided into a rule component and a lexicon component. The lexicon specifies information about the units of a language, typically words, and the rules indicate how words can be combined and manipulated. Such an organization tends to lead to a duplication of information, as can be illustrated on the basis of the sentences in (1).

C. Mahlow and M. Piotrowski (Eds.): SFCM 2009, CCIS 41, pp. 88–107, 2009.

(1) a. Die Waffenruhe hielt nur wenige Stunden.
'The truce only held for a few hours.'
b. Alphons hielt ein Buch in der Hand.
'Alphons held a book in his hand.'
c. Beatrice hielt nicht viel von ihrem Chef.
'Beatrice did not think much of her boss.'

The three sentences in (1) all contain the verb form *hielt*. In each case, it is the simple past of the verb *halten*. However, syntactically and semantically the occurrences are very different. In (1a) *halten* is intransitive, in (1b) it is transitive, and in (1c) it has both a direct object and a bound preposition *von*. These three syntactic frames correlate with three different meanings. Obviously, three different entries for *halten* are necessary, but each of them will have to state that the past tense is *hielt*. If each entry includes this information, this leads to undesirable redundancy. Following standard linguistic analysis, e.g. [1], we would like to say that there is a single lexeme *halten* with different senses. This situation is typical for a large proportion of the vocabulary and can be represented as in figure 1.

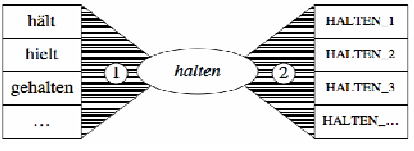

Fig. 1. The Bow-Tie Model for the lexeme *halten* with selected verb forms and senses

The central idea in WM is that it is preferable to treat the mappings 1 and 2 in figure 1 separately in NLP. In such an approach, a highly reusable component for mapping 1 can be developed, whereas the distinction between different syntactic and semantic senses in mapping 2 is left to a different component. In this context, it is worth noting that the details of the mapping in 1 are much less theory-dependent than the details of the mapping in 2, so that in mapping 1 it is much easier to foresee the type of output that will be required by client systems.

The full scope of mapping 1 does not only refer to the verb forms and the lexeme, but also to the inflectional classes involved. Whereas standard approaches, e.g. [2], use *inflectional class* to refer to a system such as that of Latin conjugations and declensions, in WM this approach is generalized so that all lexemes belong to an inflectional class. In German, two main inflectional classes can be distinguished, strong verbs including *halten* and weak verbs such as *sagen* ('say').

Apart from the inflection of lexemes, mapping 1 also encompasses word formation. In the case of *halten*, the inflectional pattern with the vowel change in the past is no longer productive, but derived verbs, e.g. *behalten* ('keep') and *zurückhalten* ('withhold'), have the same endings and the same vowel alternation as *halten*. A proper coverage of mapping 1 has to represent this relationship.

Since the 1980s, a well-known problem in NLP has been the 'lexical bottleneck'. As documented in, for instance, [3] and [4], various approaches to the reusability of lexical resources were explored to solve this problem. These approaches can be divided into

two classes, one concerned with the reuse of existing lexical resources, the other with the creation of new, flexible resources of a high quality. WM clearly falls into the latter category, because it develops new lexical resources. If existing resources are used in the development of a WM database, they are not automatically transformed into (part of) the WM database.

A less common way of classifying approaches to the lexical bottleneck is adopted in [5]. Whereas most systems and projects assume a traditional division between lexicon and rule components and concentrate on the reusability of the lexicon, WM takes a division into the two mappings of figure 1 as its starting point. As a result, the reusable resources are not so much the lexicons as such, but rather resources for the development of task-based components. This difference is also highlighted in [6]. These task-based components combine relevant parts of lexical knowledge and rules in the underlying reusable database.

The reason for preferring task-based components as the scope of reusability over full lexicons is the different type of interface adjustment that is necessary. In general, whenever a reusable resource is reused in practice, slight adjustments will have to be made to adapt it to the use in a new environment. In the more traditional approach of reusable lexicons, special morphological, syntactic, and semantic assumptions made in the client project have to be taken into account. This means that each dictionary entry has to be adapted to the theoretical decisions taken in the different rule components. If the reusable resources are WM databases, however, they contain not only morphological information about the lexical entries, but also the rules adapted to this information. The match between entries and rules means that the scope for theoretical discrepancies is much smaller. The entire component of knowledge is treated as a coherent package, so that projects using WM for their morphology can, for instance, start their analysis from lexemes. As such, the resources provided by WM are not comparable to the combination of a reusable lexicon and a morphological processor. An essential property of WM databases is that the rules and the entries are integrated in a way that will be elaborated below.

3 Rule Types

The purpose of WM is to provide all the necessary information for the mapping between word forms and lexemes, as indicated in figure 1. In operational terms, this mapping corresponds to lemmatization or morphological analysis when going from word form to lexeme or to morphological generation when going from lexeme to word form. The mapping task is divided into two main components according to whether text words, i.e. words as delimited by spaces and punctuation marks, correspond to word forms directly or not. In the latter case, text words can be split or combined. The former type of mapping is covered by WM core, whereas the latter type is performed by Phrase Manager (PM).

There are three basic rule types in WM core, IRules, WFRules, and SRules. Their syntax and use are described in detail in [7]. WM adopts a strict distinction between inflection and word formation. In linguistic theory, this distinction has been and still is controversial. Whereas [8], for instance, defends the distinction, [9] argues against it. Intuitively, the distinction is clear, at least in prototypical cases. Inflection is the adaptation of a lexeme to a particular syntactic environment, whereas word formation is the

creation of a new lexeme. When we know that *sagen* is a German verb, we expect that it has forms for the first, second, and third person singular and plural in the present and past tense. The existence of *besagen* ('mean'), *entsagen* ('renounce'), and *versagen* ('fail'), however, cannot be predicted automatically. The central difference is that inflection is paradigmatic, whereas word formation operates on a case by case basis. Formalizing this distinction, [10] proposes an operational definition with tests.

3.1 IRules: Inflectional Classes

The formalism of IRules in WM exploits the paradigmatic nature of inflection. IRules combine sets of *formatives*. Formatives correspond roughly to the linguistic concept of *morpheme*. They consist of a (possibly empty) form and a set of features, but unlike morphemes, a formative does not have to have a meaning. Formatives are the basic units combined in a word form. Combining formatives means that their forms are concatenated. (2) lists the word forms of *sagen* in the present indicative.

(2) a. sag-e (1 singular)
b. sag-st (2 singular)
c. sag-t (3 singular)
d. sag-en (1 plural)
e. sag-t (2 plural)
f. sag-en (3 plural)

The word forms in (2) are divided into formatives and accompanied by the person and number features. The statement that produces the forms in (2) is (3).

(3) `(ICat V-Stem) (ICat V-Suffix)(Mod Ind)(Temp Pres)`

The formula in (3) is part of the RIRule of weak verbs in German. It concatenates two classes of formatives and generates the word forms of the present indicative as illustrated in (2). The first class of formatives is designated by a single feature, the second class by a combination of three features. The attribute *ICat* is used in WM for all information that is necessary to designate (inflectional) formatives but that should not end up as information about the word form produced. The second formative designation in formula (3) identifies the six endings in (2). These endings constitute a class listed and fully specified in the inflection component. The first part of (3) constrains the possible stems, but does not specify their form. In the inflection component, it is an underspecified formative. The individual instantiations are specified by the lexicographer.

IRules are descriptions of inflectional classes. Their central part is the *word forms* section, which consists of a sequence of statements such as (3) generating all word forms of a lexeme. In addition, the *citation forms* section identifies one (or more) word forms as the name of the lexeme. It also consists of a statement such as (3). The *paradigms* section is optional. It can be used to group forms together independently of the formulae used to generate them. An example of a lexeme where this is useful is the present tense indicative of French regular verbs in *-ir*, e.g. *finir* ('finish'), listed in (4).

(4) a. fin-is (1 singular)
b. fin-is (2 singular)
c. fin-it (3 singular)
d. fin-iss-ons (1 plural)
e. fin-iss-ez (2 plural)
f. fin-iss-ent (3 plural)

The singular forms in (4a-c) have two formatives, whereas the plural forms in (4d-f) have a meaningless formative *-iss-* between the stem and the ending. Therefore, the best way to generate the forms in (4) is to use one statement, concatenating two formatives, for the singular and another, concatenating three formatives, for the plural. By specifying a paradigm `(Mod Ind)(Temp Pres)`, we can still retrieve all forms in (4) as a single paradigm.

There are two types of IRules, regular (RIRules) and irregular (IIRules). For RIRules, the lexicographer can add lexemes to the class. For IIRules, the linguist specifies all relevant lexemes. German weak verbs are encoded as an RIRule, whereas strong verbs such as *halten* are encoded as an IIRule.

3.2 SRules: Changes in Formatives

SRules or Spelling Rules are used in WM in all cases where the form of a formative has to be changed. They are divided into ISRules for inflection and WFSRules for word formation. Both the function and the formalism of SRules are inspired by two-level rules as used in two-level morphology, [11], [12]. The main difference between the two is that SRules can refer to the features as well as the form of a formative. In (5), the SRules for some word forms in the paradigm of *halten* are given.

(5) a. `"⟨.*[dt]⟩" (ICat V-Stem) "⟨([st].*)/e\1⟩" (ICat V-Suffix)`
b. `"⟨(.*)#A..#(.*)/\1ä\2⟩" (ICat V-Stem) (Mod Ind)(Temp Pres)(Pers 2nd)(Num SG)|(Mod Ind)(Temp Pres)(Pers 3rd)(Num SG)`
c. `"⟨(.*)#.y.#(.*)/\1ie\2⟩" (ICat V-Stem) (Temp Impf)`

The SRule in (5a) is responsible for the insertion of the *-e-* in the second person plural of the indicative *haltet*. The rule specifies two sets of formatives following each other. They are separated by a tab in the specification. The first formative in the sequence is a verb stem ending in *-d* or *-t*. It is only given as context. The second formative is specified as a verbal suffix starting with *s* or *t*. The slash in the string specification separates input and output. The backslash followed by a number copies across bracketed terms from the input, numbered from left to right. In the second formative of (5a), the output consists of *e* followed by the first bracketed term before the slash.

The SRule in (5b) is used to change the stem vowel in the second and third person singular of the present indicative, *hältst* and *hält*. One striking feature of (5b) in comparison with two-level rules is that the second formative is only indicated by means of its features. This is an advantage, because the stem vowel change is independent of the form of the ending. As indicated by (2c) and (2e), both the third person singular and the second person plural have an ending *-t*. However, whereas the former ends up

as *hält*, the latter is *haltet*. The same use of features can be observed in (5c), where the stem vowel change from *halt* to *hielt* is generally stated for the past tense.

The stem vowel change itself is stated in the first formative of (5b) and (5c). The strategy adopted here is not imposed by the WM formalism, but it is a perspicuous and effective way of stating the semi-regular alternations in strong verbs. It exploits the distinction between lexical form and surface form familiar from two-level morphology. In the lexical form, the stem vowel is replaced by a sequence of three symbols unambiguously indicating the different shapes the vowel can take. The first symbol indicates the second and third person singular of the present indicative, the second symbol the past tense and the third the past participle. These three symbols are surrounded by ##. The parts of the stem preceding and following the stem vowel are matched by the (.*). These parts are copied across by the \1 and \2 after the slash. (5b) only addresses the first symbol between ## and leaves the others unspecified. Similarly, (5c) only addressed the second symbol.

ISRules can be stated at three different levels of generality. The most specific level is that of the individual entry. An example is the rule removing the final stem consonant of *haben* ('to have') in the third person singular of the present indicative *hat*. Rules such as (5b) and (5c) are stated at the level of the IRule. They apply to all strong verbs. Rules that apply to more than one inflectional class, such as (5a), which applies both to weak and to strong verbs, are stated generally for the entire database. More general SRules apply before more specific ones. The sequence of SRule applications for *hält* is illustrated in (6).

(6) a. `"h#Aya#lt"  "t"`
    b. `"h#Aya#lt"  "et"`
    c. `"h#Aya#lt"  ""`
    d. `"hält"  ""`

(6a) represents the basic lexical form of the two formatives. In (6b), the general ISRule (5a) has applied. This is corrected by a more specific rule deleting *-et* for strong verbs with an alternation of *a/ä* or *e/i* (e.g. *gelten*, *gilt* 'be valid'), which produces (6c). Finally, the application of (5b) leads to (6d).

3.3 WFRules: Word Formation Rules

The treatment of word formation in WM is based on the assumption that word formation creates a new lexeme by applying a rule to one or more existing lexemes. Unlike inflection, this process is not paradigmatic, but applied on a case by case basis when the need arises to name a new concept. This approach is in line with classical generative theories such as word-based morphology, [13], and a-morphous morphology, [8].

Word formation processes can be divided into suffixation, prefixation, conversion, compounding, and a number of minor classes. The distinctions between these classes do not play a major role in the way they are modeled in WM. The main formal distinction is the one between WFFormatives and IFormatives. IFormatives are the formatives that exist in inflection. They provide the base to which WFRules apply. WFFormatives are formatives that are specific to word formation. These are usually affixes. In terms of this model, suffixation and prefixation can be described as

combining an IFormative and a WFFormative, compounding as combining two IFormatives, and conversion as operating on a single IFormative.

WFRules have to specify their input and their output. In (7), we find the specification of the input and output of the rule for the verb-to-verb prefixation producing *versagen* ('fail') and *behalten* ('keep').

(7) a. *source*
```
1    (WFCat V-Prefix.Non-Detachable)
(Cat V)(?IRule No-Detachable-Prefix) >
    2      (ICat V-Stem)
```
b. *target*
```
(?IRule ?)
    1 2    (ICat V-Stem)
```

The *source* section of a WFRule identifies and numbers the formatives to be combined. In (7a), the first formative is a prefix of the class including *ver-* and *be-*, the second a verb stem. (7a) excludes verb stems with detachable prefixes, because they are morphologically impossible, cf. **bezurückhalten*. By means of `?IRule`, (7a) covers stems from both RIRules (e.g. *versagen*) and IIRules (e.g. *behalten*). The *target* section has the function of specifying the order of the components in the new formative and assigning it to an IRule. In (7b), the prefix+stem combination is a new verb stem. The IRule is not directly specified by (7b), because `(?IRule ?)` refers to any IRule. However, the propagation sign > in (7a) passes the IRule of the original verb stem on to the result. Therefore, *versagen* is regular like *sagen* and *behalten* is a strong verb with the same vowel change as *halten*.

It is not difficult to see on the basis of (7) how other word formation processes can be covered in WM. Suffixation, e.g. *Haltung* ('posture'), has the WFFormative follow the IFormative. The suffix *-ung* also determines the syntactic category, the gender, and the IRule, all of which can be specified in the target. Compounding, e.g. *Gasthaus* ('inn', lit. 'guest-house'), combines two IFormatives and propagates syntactic features and IRule from the second. This will automatically produce the correct plural *Gasthäuser*. A so-called 'linking element', e.g. *-s-* in *Kriegsende* ('end of the war', lit. 'war-*s*-end'), is a separate formative. Although this is not imposed by the formalism, in the German WM database there are separate rules for each linking element. This solution is most practical in lexicographic specification. The treatment of conversion, e.g. *Tanz* ('dance$_N$') and *tanzen* ('dance$_V$'), is in line with [14] in the sense that there is no zero affix involved and the process is directional. The *-en* ending in the verb *tanzen* is an inflectional ending for the infinitive, so that the stem is the same in both words. There is therefore only a single IFormative in the source of the relevant WFRule. As a consequence, the features and IRule of the result have to be specified as properties of the target class of the WFRule.

As mentioned in section 3.2, SRules are divided into ISRules and WFSRules. The distribution of work between the two is such that WFSRules only bring about changes in the string that occur in the word formation process. There is no need to specify that *behalten* has a past tense *behielt*, because this will follow from (7). As (7) refers the resulting stem to the same IRule as *halten*, the ISRules in (5) will apply to *behalten* as well. WFSRules are only necessary for cases such as *Tänzer* ('dancer'). Here the

nominalization with the suffix *-er* triggers the umlaut. The base verb *tanzen* does not have an umlaut whereas *Tänzer* has an umlaut throughout its paradigm.

3.4 Phrase Manager

Whereas core WM rule types can be used to model mapping 1 in the Bow-Tie model in figure 1 in all cases in which word forms correspond to text words, there are also cases where there is a mismatch between word forms and text words. These cases are covered by Phrase Manager (PM), of which [15] gives a full description. PM includes mechanisms for three types of phenomena: clitics, periphrastic inflection, and phrases. In the case of clitics, a text word is split. In periphrastic inflection, two or more text words are combined as a word form of a single-word lexeme. The mechanism for phrases caters for multi-word lexemes.

In German, the most prominent case in which PM has to be used is the one of separable verbs, e.g. *zurückhalten* ('withhold'). The problems these verbs pose for the recognition of the word form and lexeme can be illustrated by the examples in (8).

(8) a. Camille hält die Informationen zurück.
'Camille withholds the information.'
b. Dorothée weiß, dass Camille die Informationen zurückhält.
'Dorothée knows that Camille withholds the information.'
c. Es ist ungerecht, diese Informationen zurückzuhalten.
'It is unfair to withhold this information.'

German has a basic word order SOV, which is found in subordinate clauses as in (8b). In main clauses, German has a verb-second rule that moves the inflected part of the verb to the position after the first constituent. As shown in (8a), this typically leaves the particle of separable verbs stranded in final position. The infinitive in German is marked by the particle *zu*, which attaches between the prefix and the main part of the separable verb, as in (8c). The most attractive analysis of the data in (8) is that they involve a lexeme *zurückhalten*, which consists of two parts that may be separated in certain syntactic contexts. This lexeme may also serve as input to word formation rules, e.g. *Zurückhaltung* ('restraint'). The formation of separable verbs is highly productive. About 47% of the almost 19,000 verbs in the WM dictionary database for German belong to this class, although the recent spelling reform, cf. [16], reduces this number slightly. Here, I will concentrate only on the use of PM in the coverage of these verbs. [17] gives a more complete analysis.

Clitic rules are used to split up a text word. Their simplest context of use is in cases such as (9), in Italian.

(9) Voglio comprarlo
'I-want to-buy-it'

When an Italian infinitive has a pronoun as its object, the two are written together. In (9), *comprarlo* has to be split into *comprare* ('buy') and *lo* ('it') before assigning each to a lexeme. The CRule performing this operation has the form of (9).

(10) `(Cat V)(Mod Inf)(Temp Pres) + (CElement Pron.Single.Cit-Form) = (CElement Pron.Single.Cit-Form), (Cat V)`

The rule in (10) states that an infinitive written together with a clitic pronoun corresponds to the clitic pronoun followed by the verb. The specification that *lo* is a second person singular pronoun is made in the list of CElements. The CRule for *zurückzuhalten* in (8c) is (10).

(11) `%separable + (CElement Conjunct.zu) + (Cat V)(Mod Inf)(Temp Pres) = (CElement Conjunct.zu), %separable + (Cat V)`

There are two main differences between (10) and (11). First, (11) not only splits the form *zurückzuhalten* into the three elements *zurück*, *zu*, and *halten*, but also recombines them into *zu* and *zurückhalten*. Second, the separable particle has a special status. The infinitive in (10) and (11) is a form arising in core WM. That is to say, when (11) is applied to *zurückzuhalten*, the system checks that there is a verb *halten*. The pronoun in (10) and *zu* in (11) are CElements. CElements are exhaustively listed in the Clitics component of PM. For the separable part of separable verbs, no such characterization exists. Whatever is left after the infinitive and *zu* have been removed will be taken to be a separable part of the verb.

For the combination of *hält* and *zurück* in (8a), two steps have to be made. First, the two elements have to be identified as belonging together. Then, they have to be combined into a single word form. The first of these steps is performed by a PIClass. A PIClass takes as its input a sentence and checks whether certain elements are found together in this sentence. It then refers these elements to a particular periphrastic inflection rule. The PIClass does not rely on parsing. It only states conditions on the morphological analysis and the order of elements in the sentence. The PIClass for separable verbs states that a verb form and a separable particle have to be found, with the verb form preceding the particle. The two elements may be separated by intervening material, e.g. “die Informationen” in (8a). The periphrastic inflection rule associated with this PIClass is (12).

(12) `(Cat V)^(Mod Inf)^(Mod Part) + %separable = (POS 1)(CFORM 2+1)(PERC 1)(Cat V)`

The rule in (12) has the same general form as the CRules in (10) and (11). In the part preceding the “=”, the elements are identified. The verb is specified as not being an infinitive or participle. The rule is only meant to apply in cases such as (8a), where a verb-second rule has moved the inflected verb form to the position after the first constituent. A relevant example where (12) does not apply because of this restriction is (13).

(13) Das Auto zu parken ist ein Problem.
‘The car to park is a problem’, i.e. Parking the car is a problem.

In (13) the infinitive *parken* (‘park$_V$’) is not combined with *ein* into a form of *einparken* (‘get into a parking space’), because when *parken* is analysed as an infinitive,

it is not in the scope of (12). The part after the "=" specifies the result. When (12) applies to (8a) after the relevant PIClass has identified *hält* and *zurück*, the two elements will be put together at the position of *hält* in the string. Their citation form is the citation form of *zurück* followed by the citation form of *halten*. The features of the resulting form are the features of *hält*, i.e. 3rd person singular present indicative.

Apart from their use in the analysis of separable verbs, CRules are also used in German for contractions such as *vom* ('of the'), analysed as *von* and *dem*. This is a closed class of items. PIClasses are also used for combinations of an auxiliary verb and an infinitive or participle, e.g. in (14).

(14) Eugen ist nach Hause gegangen.
'Eugen is to home gone', i.e. Eugen has gone home.

The PIClass for perfect auxiliaries will select *ist* and *gegangen* and hand them over to a periphrastic inflection rule that will combine them into a form of *gehen* ('go') with the person and number features and the position in the sentence of *ist*, while adding the feature `(Temp Perf)`.

PHClasses can be used to encode multi-word units as lexemes. They work in a way similar to PIClasses, but of course they do not refer to periphrastic inflection rules. They also have a more elaborate system to indicate the variability of the individual elements, the possibility of inserting elements not part of the multi-word unit, and the possibility of varying the order of the elements in the multi-word unit. For the latter, a special rule type *Transformations* has been devised. The formalism does not distinguish between idioms, collocations, formulae, and other multi-word units. The application of PHClasses raises a range of theoretical questions concerning the delimitation of lexemes from syntactic combination and the boundary between instances of and allusions to multi-word units. The PHClass mechanism has been tested for German, but no large scale encoding of German multi-word units has been undertaken.

3.5 Interaction of Rules

The best starting point for a discussion of the interaction of the rule types is the analysis of a sentence by a WM-based morphology component in which all rule types have been implemented. This is represented in figure 2.

The rule types discussed in sections 3.1-3.4 are represented as white boxes in figure 2. The small double arrows in the figure connect rule types that collaborate closely and use each other's results. The large black-and-white arrows indicate the main flow of control in analysis.

The most basic rule type is the IRule. It is impossible to use WM without using IRules, whereas all other rules can at least technically be left unspecified. If only core WM is used, the input unit is taken to be a text word. If PM rule types are specified, the input is the maximal domain in which all elements to be combined in PIClasses and PHClasses must be realized. Normally, one can assume that a sentence is the right domain, but WM does not check whether the input is a sentence, a smaller constituent, or just a sequence of words.

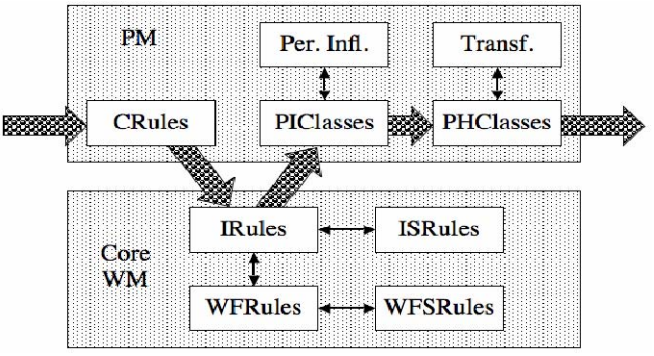

Fig. 2. Interaction of rule types in Word Manager in analysis

Every text word in the input will be assigned to one or more analyses. CRules will split text words when they apply, adding their result to the existing version. After the application of CRules including (11) to (8c), the result passed on to the IRules is (15).

(15) a. Es ist ungerecht, diese Informationen zurückzuhalten.
b. Es ist ungerecht, diese Informationen zu zurückhalten.

The application of the IRules will eliminate (15a), because *zurückzuhalten* does not exist as a word form. The IRules produce all morphological analyses of each of the word forms. The form *diese* in (15b) is analysed as the nominative or accusative of the feminine singular or the plural of *dieser* ('this'). It is not considered the task of WM to choose among these analyses, because such a choice belongs to mapping 2 in figure 1.

The application of PIClasses and PHClasses adds possible analyses, but does not eliminate any. As a consequence, the output of the analysis of (8a) will correspond to the two sequences in (16).

(16) a. Camille hält die Informationen zurück.
b. Camille zurückhält die Informationen.

For each of (16a-b), each word form will be accompanied by the set of possible inflectional analyses. On purely morphological grounds it is impossible to eliminate (16a). However, given the possibilities to eliminate the false recognition of *einparken* in (13), the existence of an analysis such as (16b) makes it highly probable that it is the right one. Therefore, it is worth presenting it as the preferred option. The possibilities of doing so depend on the way the system is used in practice.

4 Development Environment

The quality of resources depends to a large extent on the properties of the environment in which they are developed. If the formalism makes a natural way of expression possible and a variety of tools to check for consistency and correctness is available, many

errors can be avoided. Therefore, WM puts a lot of emphasis on the user-friendliness of the development environment. In the development of large morphological dictionaries, three stages can be distinguished, each of which has its own requirements.

- Linguistic specification: description of the morphological rules of a language.
- Lexicographic specification: description of the lexicon of a language in terms of the morphological rules.
- Specification of a lexical tool: deriving an operational component for a particular task from the lexical database.

For each of these development stages, WM provides separate interfaces with optimized support for the relevant tasks.

4.1 The Linguist's Interface

The task of the linguist in WM is to specify a morphological rule database for a language. A morphological rule database contains a model of the rule system that is subsequently used as a basis for lexicographic specification. This means that in specifying the rules, the linguist has to decide for each rule whether it is productive or not. If a rule is productive, it is encoded as *regular* and the lexicographer will be able to add new entries to it. All other rules are considered as closed classes, to be described exhaustively by the linguist.

It is worth elaborating what exactly it means to encode, for instance, German strong verbs as a closed class. Verbs such as *sagen* are encoded in an RIRule whereas verbs such as *halten* are encoded in an IIRule. The IIRule will contain all instances of relevant verbs as hard-coded entries. However, it need not contain *behalten* or *zurückhalten*. These verbs are not specified as entries of the IIRule covering *halten*, but as entries of the RWFRules deriving them from *halten*. They are regular as the result of prefixation, in the same way as *versagen* and *absagen* ('cancel') are derived from *sagen* in a regular way. The IIRule only has to contain all simple strong verbs, not the ones based on the application of WFRules.

The support provided in the Linguist's Interface consists of properties of three main types. The first type is related to the specification environment, the second to compilation, and the third to browsers.

The specification environment is organized as a hierarchy of windows that can be manipulated by draw-out menus and shortcut keys. At the top of the hierarchy is the *Document Window*, shown in figure 3. This window has the name of the rule database at the top and buttons for different rule types below. The buttons are divided into three parts, with general controls at the top, core WM in the middle, and PM at the bottom. Rule types that serve the classification of lexical entries are ordered in tree windows. This concerns inflection, word formation, clitics, and classes (PIClass and PHClass combined). Rule types that only serve to record changes to word forms or entries are listed in text windows. This concerns spelling rules, periphrastic inflection, and transformations changing the order of words in multi-word units.

In text windows, for instance the spelling window, rules of the type illustrated in (5) are simply typed or pasted. Each SRule has to be preceded by a unique name. As an example of a tree window, I will discuss the window for inflection, illustrated in figure 4. Other tree windows work in substantially the same way.

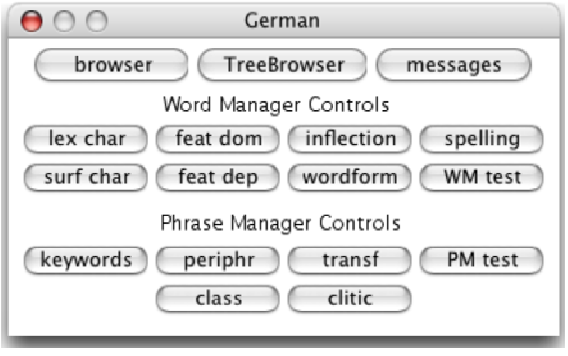

Fig. 3. WM Document Window for the German database

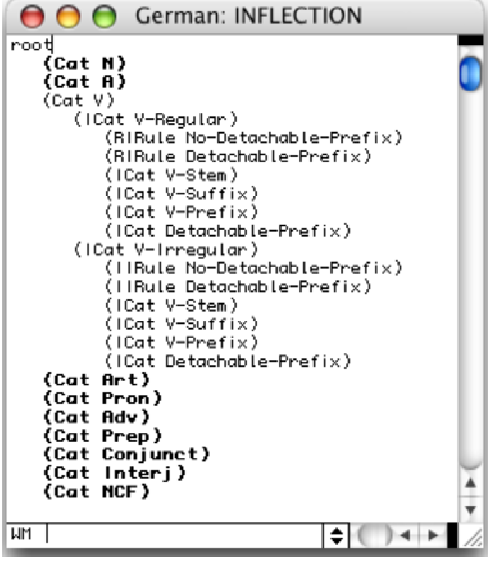

Fig. 4. Inflection window of the German database

In figure 4, The inflection window of the German database is shown. Only the part for verbs is expanded. The parts for other syntactic categories are “collapsed”, as indicated by their being in bold. A toggle command for “collapse/expand” can be selected from the Edit menu or activated by a shortcut key. There are four IRules for verbs. They are text windows in which up to five parts can be delimited. *Citation-forms* and *word-forms* consist of statements such as (3). *Paradigms* group the word forms. The optional part *ISRules* lists rule-specific spelling rules. Finally, a list of sample entries is given. For IIRules this list should be exhaustive. The full name of the IRule consists of all features of its path.

IFormatives are specified in IFormative windows. These are leaf nodes that are not specified as rules. In figure 4, V-stems and detachable prefixes are underspecified,

whereas suffixes and prefixes are fully specified IFormatives. The difference between these is only visible after opening the windows. Fully specified IFormative windows list the formatives, whereas underspecified IFormative windows only give features and leave the specification of strings to the lexicographer. As the path constitutes the full name, the names for regular and irregular suffixes are not identical. Statements such as (3) need not use the full name to refer to stems and suffixes, because they are relative to an *inflection unit*. An inflection unit is the lowest level of the tree that contains IRule(s) and IFormative windows. In figure 4, regular verbs and irregular verbs are separate inflection units.

The specification of the nodes in the tree is governed by commands such as *New Son* and *New Brother* in the Edit menu. It is also possible to cut or copy and paste nodes and to edit node names. Thus, when the inflection unit for regular verbs has been specified as in figure 4, the node `(ICat V-Regular)` can be copied and pasted as a brother to itself. This clone can then be used as a starting point for encoding irregular verbs. Only minor adjustments to the node names have to be made and a substantial part of the text of the IRule and IFormative windows can be reused.

Compilation is the process by which the specification of information in a formalism is transformed into a module that actually works. In WM, it is also used to check that the specification is correct. It is not possible, of course, to guess the intended analysis and signal each divergence from it. However, in many cases it is possible to identify whether an analysis as it is written in the formalism is plausible or not. This idea has been implemented in two ways. First, the syntactic definition of the formalism, the rules for what can be written in it, has been chosen so as to enforce consistent use. An example is a missing closing bracket or the lack of a citation form statement. Violations of the syntax result in error messages that must be solved before a working module is produced. In addition, a number of semantic constraints have been formulated. Their violation results in a warning message. Warning messages usually indicate an unintended effect, because what is produced is implausible. An example is when in a WFRule like (7), `?IRule` matches only RIRules and no IIRules. When compilation produces warnings but no error messages, the resulting module is functional.

In order to enable the compiler to check all conditions, it is necessary to apply each rule during compilation. For this reason, all rules have to have at least one example. There is no maximum of these examples. Thus, the RIRule for regular verbs specifies a number of entries that are used to illustrate WFRules. The RIRule for verbs with detachable prefix does not specify any entries directly, because it is applied when the output of the WFRule forming such complex verbs as *absagen* is referred to it.

In order to check the overall consistency of the specification, the compiler has to know which forms are allowed. This amounts to a full 'declaration' of possible forms and features. For the application of the SRules in (5), the entry for the second person plural ending in *haltet* has to specify the forms as in (17a) and the stem of *halten* as in (17b).

(17) a. `"t" "t" "et"`
b. `"h#Aya#lt" "halt" "hält" "hielt"`

In both specifications in (17), the *lexical* form is followed by a list of all permitted *surface* forms after application of SRules. In (17a), the lexical form is identical to the most common surface form, whereas in (17b) the lexical form encodes the vowel

alternation of the stem. The characters used in the lexical and surface forms of formatives are declared in the windows *lex char* and *surf char* listed in figure 3. These characters are initialised to contain all lower case characters, but umlaut characters have to be added for German, and # and *A* have to be added to *lex char* for the specification in (17b). For analogous reasons, all features are declared in *feat dom* in figure 3. Here the compiler will present the linguist with a list of undeclared attribute-value pairs and the linguist has to decide which of these are correct. In this way, it can be avoided that both `(Num SG)` and `(Num Sing)` are used in the same database. The window under *feat dep* in figure 3 offers the possibility to specify further conditions to be tested in compilation, for instance that every noun has a gender feature.

A substantial effort has been invested in making the error and warning messages given by the compiler understandable to linguists. Two main techniques have been used. The first is to open the relevant window where an error has been spotted. The second is to identify the linguistic coding problems of errors and formulate the messages in linguistic rather than computational terms.

After compilation, the rule database can be explored with three types of browser, the *general entity browser*, the *lexeme browser*, and the *tree browser*. The general entity browser can retrieve classes of rules, formatives, or entries on the basis of their type and their features. Possible queries include, for instance, to retrieve all WFRules with prefixing or all fully specified WFFormatives associated with prefixing. Depending on the type of entity it can then show entries, formatives, or rules associated with one of the entities on the list. This browser is good when the aim is to retrieve whether a particular phenomenon has been covered and which example has been chosen for it. At any point, a shortcut exists to the lexeme browser and to the relevant rule windows where the specification can be modified.

The lexeme browser gives all information related to a single lexeme L. This includes the word forms and paradigms, the creation and generation history, etc. For each word form of L the way it is built up from formatives and the application of ISRules can be shown. The creation history does the same for the application of WFRules and WFSRules building L, whereas the generation history lists the lexemes built from L. The WFCluster option represents creation and generation histories graphically.

The tree browser takes the set of entries of the database as its point of departure and offers different ways of partitioning this set. Each IRule and WFRule is considered as a class. These classes are structured in a way corresponding to the tree structure as exemplified in figure 4, so that we can also access all regular verbs or all verbs. In addition, entry features (e.g. auxiliary *haben* or *sein* for verbs, gender for nouns) and WFFormatives (affixes) can be used to delimit the target set of lexemes. These criteria can be subordinated to each other. In this way it is possible to find out, for instance, how many verbs there are and how they are distributed over word formation rules. For each class, the full list of entries can be retrieved. The tree browser is more interesting to use on a complete dictionary database.

The intended workflow in linguistic specification is to work in small, cyclic steps. One could for instance first specify an RIRule for regular verbs without detachable prefix with one example entry, together with IFormative windows for stems, suffixes and the prefix *ge-*. The next step is compiling this partial database. Then the result is inspected with the browsers, in particular the lexeme browser. If everything is correct,

the node for regular verbs can be copied to serve as a template for the irregular verbs. In this way, the rule database grows incrementally, using the existing parts as a template after validation.

4.2 The Lexicographer's Interface

The linguist and the lexicographer have clearly distinguished tasks in WM. Whereas the linguist needs a powerful formalism to state a relatively small number of rules as a description of the morphological rule system of a language, the lexicographer needs an environment in which a large number of lexical entries can be specified efficiently and consistently. The Lexicographer's Interface in WM offers such an environment by exploiting the rule database specified before. As described in more detail by [18] for English and Italian, lexicographic specification is the classification of lexical entries in terms of the rules in the rule database.

The Lexicographer's Interface assumes that a complete, compiled rule database is available. The lexicographer works with a list of words to be entered. The first decision to be made is whether the word to be entered is a simple entry or the result of a word formation rule. Let us assume we have *fühlen* ('feel') on the list. This is a simple entry which is not used as an example in the rule database, i.e. it is not *hard-coded*. Therefore, we select "Add Simple Entries" from the menu. We get a menu that first of all asks us to select an RIRule. As *fühlen* is a verb, we can restrict the list of rules to the ones for verbs and only two rules remain, the ones for verbs with and without detachable prefix. IIRules are not available in the Lexicographer's Interface.

On selecting the RIRule for verbs without detachable prefix, other parts of the specification window are highlighted. We can now ask WM to give us some example entries. If we do this, the specification of information for, for instance, *sagen* will be filled into the remaining fields. We can choose how many example entries are given and which entry is used to fill the fields. For *fühlen* we can now overwrite the information for *sagen* so that we are less likely to make errors than if we had to start with empty fields. However, for an experienced lexicographer, examples will only be necessary for difficult entries.

On completing the form, we press "Generate/Add" and WM generates a *virtual entry*. We see a lexeme browser for *fühlen* and can explore the virtual entry with the same functionality as in a lexeme browser in the Linguist's Interface. The only difference is that it has an *Add* button. We can now decide whether to add this entry or to close the browser and delete the virtual entry. The latter step will return us to the window for specifying simple entries, where we can change the values we entered the first time.

Let us now consider how to enter *behalten*. As this is a complex verb, we select a different specification window, "Add Complex Entries". This window lets us specify all the information we need to make WM generate *behalten* by the application of a WFRule. First, we are asked to select a RWFRule. As with RIRules for simple entries, we can restrict the number of rules listed, e.g. to derivation rules or to rules deriving verbs from verbs. In our case, we need the rule for non-detachable prefixes. WM now expects us to identify the formatives to be combined. It offers a pull-out menu with all relevant WFFormatives for this rule, i.e. all non-detachable verbal prefixes, from which we select *be*. For the stem, we can either enter it manually or ask

WM to search for *halten* and select the result. After specifying all source formatives and the RWFRule, we can now press "Generate Target" to produce the stem of the new lexeme. This stem has the form `°beh#Aya#lt`. This may seem daunting, but as lexicographers we do not have to interpret or manipulate it. We can simply press "Generate/Add" so that WM will produce a lexeme browser for a virtual entry *behalten*. On the basis of this information on what the entry would be like, we can then decide whether to add it or not.

As illustrated by these examples, the Lexicographer's Interface activates the rules in the rule database on the basis of information specified by the lexicographer to generate new entries. This approach to lexicographic specification has the advantage that all new entries are possible words in view of the existing entries and rules of the language. The way the information is presented means that a detailed understanding of the rules and their operation is not necessary for lexicographic specification.

A detail that has been left out of the above description is that the German WM rule database states the feature dependency that a verb needs to specify its auxiliary. In the case of *fühlen*, this can be taken over from the example entry selected as a template. For *behalten* it must be specified manually. In the development of WM the complications of calculating this feature were judged not to outweigh the burden of manual specification. If the feature is missing, pressing the "Generate/Add" button will produce an error message asking for its specification. The Lexicographer's Interface also allows for the specification of optional entry features.

Apart from the addition of features, the Lexicographer's Interface also offers the possibility of specifying entry-specific ISRule and WFSRules. This makes it possible to adjust forms where necessary. What is not possible in the Lexicographer's Interface, however, is to change what is specified in the rule database. This entails that if an error is detected in a rule, or if a rule was forgotten in the linguistic specification, or if there is an error in a hard-coded entry used as an example for the benefit of the compiler, the lexicographer cannot correct this. The entries created in the Lexicographer's Interface have to be exported, the rule database modified and compiled, and the exported entries re-imported to amend any rules. This procedure is the price to be paid for the quality ensurance and support offered by the general setup of the system.

4.3 The Developer's Interface

A WM dictionary database can be a useful device in, for instance, linguistic research, in particular in combination with the tree browser. However, it is not primarily intended to be used as a stand-alone device. In line with the distribution of labour illustrated in figure 1, the idea is to produce operational components, so-called *lexical tools*, based on the WM dictionary database. A lexical tool is a dedicated, self-contained Finite-State Transducer (FST). It performs a particular task for which all the relevant information is taken from the WM dictionary database and encoded in the FST. Whereas the WM dictionary database is large and flexible in the information it contains, but bound to a restricted software and hardware environment, lexical tools derived from it are small, efficient, and can be generated for any platform and environment.

The derivation of a lexical tool is an operation that is only performed once for a particular task and WM dictionary database. The interface to enter the specifications for the FST consists of an algorithm with documentation that indicates which type of

information has to be entered where. It is meant to be used by specialist software engineers. [19] and [20] describe this in more detail.

Generic applications for which lexical tools have been developed are lemmatization, morphological analysis, and morphological generation. In lemmatization, the input is an inflected word form and the output is the lexeme(s) the word form can belong to. Morphological analysis also returns the features of the word form in the lexeme. The analysis can be purely inflectional or also involve word formation rules for relevant lexemes. Morphological generation starts from a lexeme and returns the set of word forms and/or derived lexemes.

An interesting application is the *generative spell checker*. This tool works as a normal spell checker as far as words are documented in the WM database. For non-found words, it tries to produce them by applying word formation rules. As compounds in German are written together, this procedure eliminates a substantial part of the overflagging of normal German spell checkers.

4.4 Evaluation of the Development Environment

The structure of the development environment is based on the design philosophy of WM, which can be summarized as making as few expressive options available as necessary. The split between the Linguist's Interface and the Lexicographer's Interface is founded on the assumption that the two types of specification are different in crucial respects. A WM linguist works meticulously on a fairly narrow set of rules, whereas a WM lexicographer has to work through large numbers of entries efficiently. It is not normally expected that these tasks are performed by the same person. Writing and editing rules requires a much deeper understanding of the morphology of the language and of the WM formalism.

There are a number of advantages of this split. It is easier to specify lexical entries, and much of the existing resources in WM would probably not have existed if the people specifying entries would have to learn the entire rule formalism. It is also easier to avoid inconsistencies. More than one person can work in the specification of lexical entries, but if they had the power to change rules this would be extremely dangerous. In the development of the English and Italian lexicon databases, described in [18], the organization of the work was such that changes to the rules occurred very rarely and did not result in the loss of much time. Therefore, in the perception of the people working in this project, the slight inconvenience of having to wait with the encoding of certain entries was clearly outweighed by the substantial advantage of being able to specify entries with so little effort while maintaining linguistic consistency.

5 Available Resources and Their Use

The general philosophy underlying WM is that it is better to invest a lot of effort in a high-quality resource than to produce a poor resource on-the-cheap. This is reflected in the nature of the development process, which involves rule databases, dictionary databases, and lexical tools.

Rule databases for core WM have been developed for German, Italian, and English. Of these, the Italian one is well-documented in [21]. It can stand as an example

for the general approach. Subsequent changes have occurred in particular in the treatment of neo-classical word formation, cf. [22], and in exocentric and synthetic compounds, cf. [23] and [24]. In addition, the mechanism for separable verbs in German is documented in [17]. This involves PM components as well. A more complete PM rule database for English is described by [25], but it is not fully compatible with later changes to the English core WM rule database. The rule databases are not available as such, because their main purpose is to serve as input for dictionary databases.

Dictionary databases have been developed for German, Italian, and English. The German database contains over 230000 lexemes. The Italian and English databases, whose development is documented in [18], contain approximately 50000 lexemes each. A number of free applications give access to these databases, though not in a way that they can be modified or extended. Access is through the use of lexical tools derived from the dictionary databases.

Lexical tools have been used to produce the information available on www.canoo.net. This information is useful for learners of German or anyone with morphological queries. Another use of lexical tools is in the ELDIT language learning environment, described in [26]. The integration of lexical tools in research projects such as these is arguably the most typical use of WM. However, there are also commercial users, including Google, *Neue Zürcher Zeitung*, and Zingarelli, who bought licences to use WM resources in information retrieval, hyphenation. and other tasks. An important advantage of WM lexical tools is that they can cover all of morphology, including periphrastic forms and particle verbs, in a single tool.

References

1. Matthews, P.H.: Morphology: An Introduction to the Theory of Word Structure. Cambridge University Press, Cambridge (1974)
2. Aronoff, M.H.: Morphology by Itself: Stems and Inflectional Classes. MIT Press, Cambridge (1994)
3. Atkins, B.T.S., Zampolli, A. (eds.): Computational Approaches to the Lexicon. Clarendon, Oxford (1994)
4. Walker, D.E., Zampolli, A., Calzolari, N. (eds.): Automating the Lexicon: Research and Practice in a Multilingual Environment. Oxford University Press, Oxford (1995)
5. ten Hacken, P.: Two Perspectives on the Reusability of Lexical Resources. McGill Working Papers in Linguistics 14, 39–49 (1999)
6. ten Hacken, P.: 'Representing Computational Dictionaries in Dedicated Formalisms. DATR, Word Manager'. In: Gouws, R.H., Heid, U., Schweickard, W., Wiegand, H.E. (eds.) Dictionaries. An International Encyclopedia of Lexicography. Mouton de Gruyter, Berlin (to appear)
7. Domenig, M., ten Hacken, P.: Word Manager: A System for Morphological Dictionaries. Olms, Hildesheim (1992)
8. Anderson, S.R.: A-Morphous Morphology. Cambridge University Press, Cambridge (1992)
9. Booij, G.: Against Split Morphology. In: Booij, G., van Marle, J. (eds.) Yearbook of Morphology 1993, pp. 27–49. Kluwer, Dordrecht (1993)
10. ten Hacken, P.: Defining Morphology: A Principled Approach to Determining the Boundaries of Compounding, Derivation, and Inflection. Olms, Hildesheim (1994)

11. Koskenniemi, K.: Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production. University of Helsinki, Department of General Linguistics Publications No. 11 (1983)
12. Karttunen, L. (ed.): KIMMO: A Two Level Morphological Analyzer. Texas Linguistic Forum 22, 163–278 (1983)
13. Aronoff, M.H.: Word Formation in Generative Grammar. MIT Press, Cambridge (1976)
14. Don, J.: Morphological Conversion. LEd & OTS, Utrecht (1993)
15. Pedrazzini, S.: Phrase Manager: A system for Phrasal and Idiomatic Dictionaries. Olms, Hildesheim (1994)
16. Johnson, S.: Spelling Trouble? Language, Ideology and the Reform of German Orthography. Multilingual Matters, Clevedon (2005)
17. ten Hacken, P., Bopp, S.: Separable Verbs in a Morphological Dictionary for German. In: Coling – ACL 1998: Proceedings of the Conference, Université de Montréal, pp. 471–475 (1998)
18. ten Hacken, P.: Word Formation and the Validation of Lexical Resources. In: González Rodríguez, M., Paz Suárez Araujo, C. (eds.) LREC 2002: Third International Conference on Language Resources and Evaluation – Proceedings, pp. 935–942 (2002)
19. Pedrazzini, S., ten Hacken, P.: Centralized Lexeme Management and Distributed Dictionary Use in Word Manager. In: Schröder, B., Lenders, W., Hess, W., Portele, T. (eds.) Computers, Linguistics and Phonetics between Language and Speech, Proceedings of the 4th Conference on NLP, Konvens 1998, Bonn, Germany, pp. 365–370. Lang, Frankfurt am Main (1998)
20. Pedrazzini, S.: The Finite State Automata's Design Patterns. In: Champarnaud, J.-M., Maurel, D., Ziadi, D. (eds.) WIA 1998. LNCS, vol. 1660, pp. 213–219. Springer, Heidelberg (1999)
21. Bopp, S.: Computerimplementation der italienischen Flexions- und Wortbildungsmorphologie. Olms, Hildesheim (1993)
22. Petropoulou, E., ten Hacken, P.: Neoclassical word formation in WM electronic dictionaries. In: Braasch, A., Povlsen, C. (eds.) Proceedings of the Tenth Euralex International Congress, Copenhagen, Denmark, August 13–17, pp. 169–174 (2002)
23. ten Hacken, P.: Phrasal Elements as Parts of Words. In: Hajičová, E., Kotesovcova, A., Mirosky, J. (eds.) Proceedings of CIL17, CD-ROM. Matfyzpress, Prague (2003)
24. ten Hacken, P.: Phrases in Words. In: Tschichold, C. (ed.) English Core Linguistics, pp. 185–203. Lang, Bern (2003)
25. Tschichold, C.: Multi-Word Units in Natural Language Processing. Olms, Hildesheim (2000)
26. ten Hacken, P., Abel, A., Knapp, J.: Word Formation in an Electronic Learners' Dictionary. International Journal of Lexicography 19, 243–256 (2006)

Morphological Analysis Using Linguistically Motivated Decomposition of Unknown Words

Stephan Bopp and Sandro Pedrazzini

Canoo Engineering AG, Kirschgartenstrasse 5,
4051 Basel, Switzerland
{stephan.bopp,sandro.pedrazzini}@canoo.com

Abstract. Integrating the decomposition of unknown morphologically complex words can enhance the recognition rates of morphological analyzers. Using linguistically motivated strategies for this decomposition leads to even more expressive results. The approach described here uses word formation rules and filtering techniques to analyze and decompose words that are not contained in the underlying dictionary database. The average recognition rate of our German analyzers, applied to our test corpus, increased from 91% to 95,4%. Together with the current implementation, further future decomposition strategies will be presented.

Keywords: Morphological analysis, controlled word decomposition, finite state tools.

1 Introduction

The load of electronically available textual information that has to be treated in one way or another is still growing rapidly. Finding the relevant information in large corpuses and knowledge bases is costly both in terms of time and resources. Every step that helps to automate such tasks provides a significant reduction of costs. In a text, a search term may occur in many different forms, e.g., in regional spelling, as an inflected word form, etc. That is why their normalization requires sophisticated morphological techniques such as recognition of spelling variants and mapping of inflected forms onto their respective base forms.

It is crucial that such techniques are based on large dictionaries – the quality of the analysis is directly proportional to the size and quality of the dictionary. But however large a dictionary database may be, in languages like German new words are created every single day through word composition and word derivation. Therefore, we do not only need large dictionaries, but also rules that cover these highly productive word formation processes (see [8]). We need tools for the decomposition of unknown complex words into known word elements. By "unknown word" we mean a word form not contained in the underlying dictionary database, i.e., a word form that cannot be recognized directly.

If these tools provide additional grammatical information, which, in a language like German, is deducible from the known base components, the quality of the morphological analysis can be enhanced even further. This paper describes morphological

C. Mahlow and M. Piotrowski (Eds.): SFCM 2009, CCIS 41, pp. 108–117, 2009.

tools that handle the recognition of words that are not contained in the dictionary by using techniques of controlled word decomposition.

1.1 Word Manager Dictionary Database

The underlying dictionary database is Word Manager (WM), a system for the specification, use and maintenance of morphological dictionary databases as described in [4] and [6]. A WM database describes the morphology of a language by means of comprehensive sets of rules for inflection and word formation (and spelling). Dictionary entries are specified by assigning a word stem to an inflection rule that generates their entire inflectional paradigm. If they are morphologically complex, they are assigned to a word formation rule. Word formation rules create new lexemes by combining existing lexemes with each other or with word formation affixes into a new stem, which is then assigned to a specific inflection rule. Therefore, for each lexeme contained in a WM database all inflected word forms are known and, if applicable, also how it has been created.

The WM database for German currently contains more than 250.000 entries. 89% of these entries are morphologically complex, i.e., composed of or derived from the 11% of base entries. This illustrates not only the highly generative character of the German language but also the potential of exploiting word formation rules.

1.2 WMTrans Finite State Transducers

To deploy WM dictionary databases within applications that need morphological information, the lexicon entries are compiled into finite state transducers (WMTrans). The production of WMTrans Finite State Transducers for Word Manager data is carried out by the WMTrans Finite State Transducer Framework: based on a specification of the target transducer's input and output, the corresponding transducer is generated automatically. A whole range of WMTrans products have been and can be developed: recognizers, lemmatizers, analyzers and generators for inflection and word formation, and combinations of them, for different vocabulary sizes.

2 Analysis of Unknown Words through Controlled Decomposition

2.1 The Unknown Word Transducers

The Unknown Word Transducers are derived from the WM dictionary database described above. The word formation rules used in WM to create new entries are, so to speak, applied reversely for the analysis of unknown words: Since these word formation rules define how word stems and affixes may be combined into new words, they can also be used to split up unknown strings into grammatically correct combinations of known stems and affixes.

However, if one simply applies the whole set of WM word formation rules with all WM entries, the analysis of unknown words tends to result in far to many, and worse, inadmissible decompositions. Within the framework of the Unknown Word Transducers, the analysis is carried out in a controlled environment containing the following elements:

- a set of word formation rules and affixes,
- a set of admissible base words,
- a set of postfilters for the elimination of incorrect analyses.

These elements are applied in the following order:

- recognition of known words, i.e., words contained in the WM dictionary
- application of the word formation rules with the set of admissible base words
- application of postfilters

Figure 1 illustrates the data generation for the Unknown Word Analyzer and the text processing at runtime.

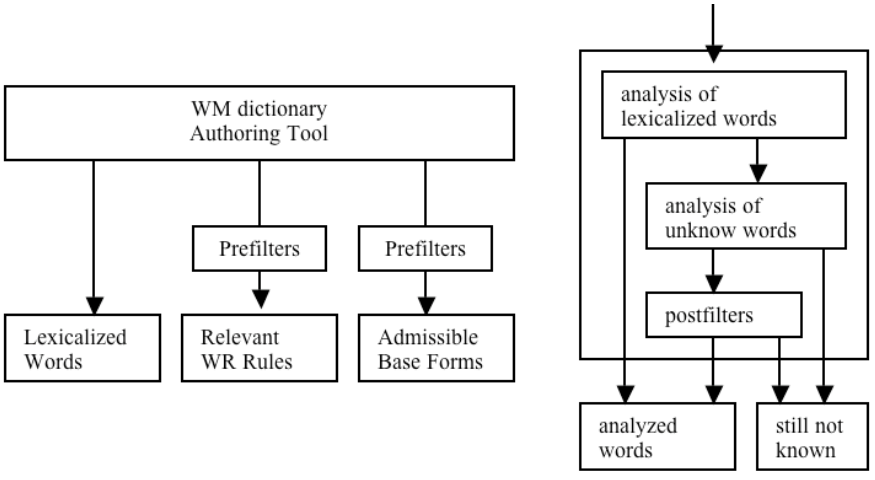

Fig. 1. Data Generation for the Unknown Word Analyzers *(left)* and runtime analysis *(right)*

2.2 Defining the Set of Word Formation Rules and Affixes

The generation history of the more than 250.000 entries in the WM dictionary as well as a large text corpus analysis were at the basis of the selection of the most relevant word formation rules and affixes. Out of almost 500 rules (counting each affix as one rule) only about 150 were chosen:

- Unproductive rules and affixes forming only few words were excluded, assuming that no (or hardly any) new words are likely to be formed with them; e.g., the noun-forming suffix *e* (*Stärke, Schwäche, Härte, Größe*).
- Some rules and affixes were excluded because their rate of undesired analyses is much higher than their rate of correct results. Examples include the adjective forming suffix *-[e]n* (*golden, silbern, graniten*) and all word formation rules for conversion (zero-derivation; e.g., adjective-to-verb conversion: *dicht/dichten, sicher/sichern*).

- Word formation rules are not allowed to be applied recursively, i.e., only one word formation rule may be applied for the segmentation of a string. Exceptions are some of the most productive combinations of word formation rules. For example, it is allowed to apply two rules for German noun+noun-compounding for the segmentation of a string: "Hausbeleuchtungsmonteur" = [[Haus+Beleuchtung] +s+Monteur].

2.3 Defining the Set of Admissible Base Words

The set of admissible word forms contains all strings that may be used for the segmentation of unknown strings. Excluding irrelevant base words on different levels can reduce the number of undesired analyses:

- Exclusion of word classes: only nouns, adjectives, verbs and adverbs are allowed, i.e., all other word classes are excluded.
- Exclusion of lexemes: entire lexemes with all their inflected word forms are excluded, e.g., the uncommon noun "Lichen" (*lichen*, in the sense of a disease) because it is homonymous with inflected forms of the diminutive suffix *-lich*.
- Exclusion of word forms: specific word forms are excluded, e.g., certain stems of irregular verbs which are homonymous with nouns: "schloss", "spross", "zwang".
- Exclusion of graphical forms: all two and three letter words are excluded except for the most common ones like e.g., "Ei", "Öl", "See", "neu".

2.4 Deleting Undesired Results

The introduction of the prefilters described above reduces significantly the number of inadmissible decompositions, but the restricted sets of relevant Word Formation rules and admissible base words still produce some wrong and undesired results. Many of them are intercepted and deleted by postfilters. By means of the result string, the word formation rule applied, and other information, a postfilter defines which (types of) results are cancelled from the list of analyses.

Example 1: German nouns ending in *-heit* are always followed by the linking element *s* when in first position of a compound. A postfilter deletes all analyses that do not meet this condition. Therefore, only the first result of the analysis of *Gesundheitsorte* is passed on to the final list of results: 1) "Gesundheit+s+orte"; 2) "Gesundheit+sorte".

Example 2: The adverb *gratis* may only be combined with nouns. A postfilter only accepts compounds with *gratis* in first position if the word in second position is a noun: "gratis+lieferung" (*free delivery*) is accepted, "gratis+geliefert" *(delivered free of charge*) is rejected. The latter must be written in two words. In this manner, postfilters can also be used for spell checking of unknown words. See paragraph 3.2.

2.5 Results of the Analysis

It is possible to customize the output of the Unknown Word Transducers to the needs of specific client applications. It contains any, a combination, or all of the following types of information for every analyzed word:

- citation form (infinitive of verbs, nominative singular of nouns, etc.)
- word class (noun, adjective, verb, etc.)
- word form features (number, case, person, etc.)
- inflection class
- segmentation (the segments are separated by a + sign)
- word formation rule applied
- citation form and word class of the base lexemes used for the segmentation

Table 1. Possible analyzer output for query "Gesundheitsortes" (*health place*, i.e., *health spa*)

Query:	Gesundheitsortes
Cit-Form:	Gesundheitsort
Grammatical Features:	(Cat N)(Gender M)(Num Sing)(Case Genitive)
Inflection Rule:	RIRule N-Regular es/e
Segmentation:	gesunheit+s+ortes
Word Formation Rule:	WFRule Compounding N-Compound N+s+N
Base Elements:	Gesundheit (Cat N)
	s (WFCat Linking-Element)
	Ort (Cat N)

Note that the results of this controlled decomposition of non-lexicalized words contain more than "only" the citation form and the word-class of the analyzed word plus an indication of the base elements. They also provide detailed information on the inflection class of the analyzed lexeme, which word formation process has formed it, and the grammatical features of the initially unknown word form encountered in the input text.

The inflection information delivered as a part of the result of the analysis also allows to generate the whole inflection paradigm of the recognized word.

3 Current Implementation of the Unknown Word Analyzers

The implementation of the Unknown Word products has been realized through a combination of finite state machines (FSMs). The information needed is derivable from the WM database. It consists of:

- complete inflection paradigms for lexicalized words, i.e., all inflected word forms of the words contained in the WM database,
- information about stems, endings and orthographical surface variants,
- word formation elements such as affixes and linking elements,
- word formation rules describing the derivation and word formation processes for German, English and Italian.

3.1 The Finite State Components

The information mentioned above is translated into optimized regular expressions and then distributed to three finite state components:

- **Inflection transducer:** This is the tool for the analysis of known word. It ensures high processing speed by avoiding unnecessary segmentation and analysis. It also maps the segments detected during the unknown word analysis to lexicalized words.
- **Surface finite state automaton:** This automaton codes all possible word elements specified in the WM dictionary. It contains all stems with their orthographical variants as well as all word formation elements (affixes, linking elements). It segments an unknown string and then outputs a list of all possible segmentations.
- **Word formation rules transducer:** This transducer handles the information on word formation rules. It matches the segmentations delivered by the surface finite state automaton to a specific word formation rule. By means of these rules is also computes specific features of the analyzed word, i.e., its word class for the Unknown Word Lemmatizer, its grammatical features for the more sophisticated Unknown Word Analyzer, as well as the new citation form.

These finite state components are integrated in the Unknown Word applications. The output of the surface finite state automaton is used as input for the analysis of the word formation rules transducer.

In some cases, the output of the surface finite state automaton consists of a long list of segmentations. In order to reduce the set of segmentations and to improve the performance of the process, some heuristics are applied before the segmentations are passed on to the word formation rule transducer. One heuristic determines that the segmentation with the smallest number of segments is the most likely one to be correct. For example:

Of the two detected segmentations of *Gründungskapital* only the one with fewer segments is passed on for further analysis: "gründung+s+kapital" (*initial capital* literally *foundation capital)* vs. "grün+dung+s+capital" (*green manure capital*).

Exceptions to this general heuristic are frequent with compounds containing linking elements. These exceptions are handled by another heuristic:

Both of the segmentations of *Examenstage* are possible words and therefore passed on for further analysis: "examen+stage" (*examination apprenticeship)* and "examen+s+tage" (*examination days*).

3.2 Processing Levels

Two different processing levels handle the recognition of unknown words:

Data Generation Level: As shown above, the word formation data is generated by means of the Word Manager authoring tool that allows us to tune the word formation rules and base elements needed for the Unknown Word Analyzer. Fine-tuning is

based on cycles of rule selection and analysis of a text corpus of several millions of words. Through prefiltering (see paragraphs 2.2 and 2.3), many elements that are irrelevant for unknown word recognition can be identified and eliminated.

Runtime Level: Undesired decompositions occurring in spite of the prefiltering are eliminated by postfilters (see paragraph 2.4) at runtime. The framework provides a full range of possibilities for the definition of filters necessary to maximize the precision of word analysis:

- On the level of the segmentation, defined as regular expression. Example: Delete all analyses with the segmentation "*ung + los" (nouns ending in *-ung* obligatorily take *s* when combined with the suffix *–los.* See below for a more precise filter definition)
- On the level of the word formation rule. Example: Delete all results with (WFRule Adv+Adj) and the segmentation "gratis + *".
- On the single component level. Example: Delete all results "*ung + los", if the first element is a noun, has a citation form "*ung", and inflects with the inflection class -/e.

It is also possible to use postfilters for the spell checking of unknown words. We use postfilters in particular for composed and derived words that were affected by the German spelling reform. The filters deal with cases like words written in one or two words (old: spazierengehen – new: spazieren gehen) and orthographic changes related to word formation (old: Schne**ll**ieferung – new: Schne**lll**ieferung). In these cases, the postfilters add relevant spelling information to the matching results instead of simply deleting them. The postfilters are specified in XML and read at load time.

Table 2. Sample Analyses of complex words using the Unknown Word Lemmatizer and Analyzer

Query	Result using WMTrans Unknown Word **Lemmatizer**	Result using WMTrans Unknown Word **Analyzer**
hingekrochen	Cit-Form: hinkriechen Class: verb	Cit-Form: hinkriechen Class: verb Form: past participle Segmentation: hin+gekrochen WFRule: Derivation.V-To-V.Prefixing hin = prefix kriechen = verb
aggressionsfreieres	Cit-Form: aggressionsfrei Class: adjective	Cit-Form: aggressionsfrei Class: adjective Form: degree comparative, -es Segmentation: aggression+s+freieres WFRule: Compounding. N+A.N+s+A Aggression = noun s = linking element frei = adjective

3.3 Unknown Word Products and Their Output

We have developed three different types of products for the analysis of unknown Words: Recognizers, Lemmatizers and Analyzers. A public Java API allows the integration of all these products into other programs.

- The Unknown Word Recognizer indicates whether or not an input string is a known word form or a possible correct *ad hoc* word form.
- The Unknown Word Lemmatizer provides additional information on citation form and word class.

The Unknown Word Analyzer delivers a full range of (parameterizable) information on grammatical features, inflection class, word formation rule applied, segmentation, base words with their word class(es).

3.4 Recognition Rates

We used our Unknown Word Lemmatizer for the analysis of a test corpus containing several millions of words. The test corpus includes a broad range of text types, i.e., fiction, newspaper texts, scientific and technical documents. The current version of the Unknown Word Lemmatizer recognizes an average of 95.4% of words occurring in the test corpus. Most of the words belonging to the 4,6% of unknown words are proper names, foreign words in quotes and uncommon technical terms. While the results are significantly above average for common descriptive writings, they are lower for technical reports and short newspaper articles containing more proper names and foreign language quotes.

Table 3 shows a comparison between the WMTrans Lemmatizer, recognizing only lexicalized words, and the WMTrans Unknown Word Lemmatizer, which recognizes both lexicalized and non-lexicalized words. The table shows that the range of unknown words can be reduced by up to 77% if controlled word decomposition is applied.

Table 3. Remaining Unrecognized Words [not] using Unknown Word Recognition[1]

Text Type	WMTrans Lemmatizer	WMTrans Unknown Word Lemmatizer
Avantgarde	7.3%	4.4%
Die Zeit	7.5%	4.4%
Science 1	16.5%	3.8%
Science 2	14.7%	6.3%
NZZ	9.1%	3.5%
Medical	17.2%	8.1%
Der Spiegel	10.3%	4.8%
Vop	6.4%	1.8%

[1] The percentage of unknown words includes proper names, foreign words, etc.

4 Further Steps

We plan further recognition strategies for our Unknown Word Analyzers, always considering the important trade-off between improvement of recognition rate and avoidance of over-recognition.

In the current version, if one segment is not recognized, the whole compound word is rejected (not recognized). There are, however, special cases where it is clear which parts are known and which are not known: words written with a hyphen like the word *Schönheits-Contest* (considering *Contest* as unknown). Here, we could enhance the Unknown Word Analyzers in such a way that it is able to analyze this type of "semi-known" compounds and deliver a meaningful result.

Recognizing such an element when written in one word without a hyphen has turned out to be a big cause of over-recognition. Some heuristics in this context are needed and are currently under study.

Table 4. Analysis of words with hyphen, with an unknown segment

Query	Explanation	Analysis
Schönheits-Contest	Only the first element is recognized, the second is considered unknown	Schönheits-Contest Schönheit = (Cat N)... s = Linking-Element Contest = unknown Schönheits-Contest = Unknown Compound (with Element "Schönheit")
Contest-Gewinners	The second element is known, the first element is unknown	Contest = unknown Gewinner = (Cat N)(Gender M)(Num SG)(Case Genitive)(CompElement 2) Contest-Gewinners = (Cat N)(Gender M)(Num SG)(Case Genitive)

5 Conclusion

This paper has shown that morphological analysis of German can be significantly improved by the integration of tools for the decomposition of unknown words based on word formation rules and filtering techniques. Not only the number of recognized words increases considerably, but also the quality of the information that can be provided for initially "unknown" words. An interesting added feature consists in the possibility to generate the entire paradigm of the recognized "unknown" words.

The analyzers are available as WMTrans products that support most of the common platforms. The product range includes morphological components for English and Italian. WMTrans products are being used by private enterprises as well as public institutions in diverse domains such as content managing, product cataloguing, construction of electronic dictionaries (e.g., [9]), references in online dictionaries (e.g., [3, 7, 11]), and e-learning projects (e.g., [1, 5]). The search function in the online German language site Canoo.net ([2]) is another application that is based on WMTrans Unknown Word products.

References

1. Abel, A.: ELDIT (Elektronisches Lernerwörterbuch Deutsch-Italienisch) und elexiko: Ein Vergleich. In: Klosa, A. (ed.) Lexikografische Portale im Internet (= OPAL-Sonderheft 1/2008, hrsg. vom Institut für Deutsche Sprache Mannheim), pp. 175–189. Mannheim (2008)
2. Canoo.net: German dictionaries and grammar, http://www.canoo.net
3. DIX: Deutsch-Spanisch Wörterbuch, http://dix.osola.com/
4. Domenig, M., ten Hacken, P.: Word Manager: A System for Morphological Dictionaries. Georg Olms Verlag, Hildesheim (1992)
5. Eldit: http://www.eurac.edu/eldit
6. ten Hacken, P., Domenig, M.: Reusable Dictionaries for NLP: The Word Manager Approach. Lexicology 2, 232–255 (1996)
7. Leo, http://dict.leo.org
8. Lüdeling, A., Fitschen, A.: An Integrated Lexicon for the Analysis of Complex Words. In: Proceedings of EURALEX 2002, Copenhagen (2002)
9. OWID: *elexico*, http://www.owid.de/elexiko_/index.html
10. Pedrazzini, S.: Periphrastic Inflection Clustering for Term Extraction. In: Proceedings of the Seventh International Symposium on Communication and Applied Linguistics, Editorial Oriente, Santiago de Cuba (2001)
11. Pons: Das Online-Wörterbuch in fünf Sprachen, http://www.pons.eu

Corpus-Based Lexeme Ranking for Morphological Guessers

Krister Linden and Jussi Tuovila

University of Helsinki, Helsinki, Finland

Abstract. Language software applications encounter new words, e.g., acronyms, technical terminology, loan words, names or compounds of such words. To add new words to a morphological lexicon, we need to determine their base form and indicate their inflectional paradigm. A base form and a paradigm define a lexeme. In this article, we evaluate a lexicon-based method augmented with data from a corpus or the internet for generating and ranking lexeme suggestions for new words. As an entry generator often produces numerous suggestions, it is important that the best suggestions be among the first few, otherwise it may become more efficient to create the entries by hand. By generating lexeme suggestions with an entry generator and then further generating some key word forms for the lexemes, we can find support for the lexemes in a corpus. Our ranking methods have 56–79% average precision and 78–89% recall among the top 6 candidates, i.e., an F-score of 65–84%, indicating that the first correct entry suggestion is on the average found as the second or third candidate. The corpus-based ranking methods were found to be significant in practice as they save time for the lexicographer by increasing recall with 7–8% among the top candidates.

1 Introduction

New words are constantly finding their way into daily language use. This is particularly prominent in rapidly developing domains such as biomedicine and technology. The new words are typically acronyms, technical terminology, loan words, names or compounds of such words. They are likely to be unknown by most hand-made morphological analyzers. In many applications, hand-made guessers are used for covering the low-frequency vocabulary or the strings are simply added as such.

Mikheev [1,2] pointed out that words unknown to the lexicon present a substantial problem for part-of-speech tagging, and he presented a very effective supervised method for inducing English guessers from a lexicon and an independent training corpus. Oflazer et al. [3] presented an interactive method for learning morphologies and pointed out that an important issue in the wholesale acquisition of open-class items is that of determining which paradigm a given citation form belongs to.

Recently, unsupervised acquisition of morphologies from scratch has been studied as a general problem of morphology induction in order to automate the morphology

C. Mahlow and M. Piotrowski (Eds.): SFCM 2009, CCIS 41, pp. 118–135, 2009.

building procedure. For overviews, see Wicentowski [4] and Goldsmith [5]. If we do not need a full analysis, but only wish to segment the words into morph-like units, we can use segmentation methods like Morfessor [6]. For a comparison of some recent successful segmentation methods, see the Morpho Challenge [7].

Although unsupervised methods have some advantages for less-studied languages, for the well-established languages, we have access to fair amounts of lexical training material in the form of analyses in the context of more frequent words. Especially for Germanic and Finno-Ugric languages, new words tend to be compounds of acronyms and loan words with existing words. For these languages, there are already large-vocabulary descriptions available. In English, compound words are written separately or the junction is indicated with a hyphen, but in other Germanic languages and in the Finno-Ugric languages, there is usually no word boundary indicator within a compound word. It has previously been demonstrated by Lindén [8] that already training sets as small as 5000 word forms and their manually determined base forms will give a reasonable result for guessing base forms of new words by analogy. The experiments were performed on a set of languages representing different language families, i.e., English, Finnish, Swedish and Swahili.

In addition, there are a host of large but shallow hand-made morphological descriptions available, e.g., the Ispell collection of dictionaries [9] for spell-checking purposes, and many well-documented morphological analyzers are commercially available, e.g., [10]. It has also been demonstrated by Lindén [11] that there is a simple but efficient way to derive an entry generator from a full-scale morphological analyzer implemented as a finite-state transducer. Such an entry generator can be used as a baseline for more advanced entry guessing methods.

Using the entry generator developed by Lindén [11], we can generate lexeme candidates, i.e., base form and paradigm combinations, for new words with the entry generator and then further generate key word forms for the lexeme candidates. Using these lexeme candidates with key word forms, a person with native skills can select the correct entry for a new word. With this method, we encoded a set of words based on an open source dictionary project with a different encoding scheme than ours[1]. We selected all the words that were unknown to our lexicon and used the entry generator to encode the new words according to the guidelines of *The Research Institute for the Languages of Finland* [12] used in our lexicon. The reclassification took approximately 20 hours of work during which a list of 11026 new entries was created. As the words had been categorized, we were able to take advantage of the existing categories to guide the process, but a number of systematic mismatches and ambiguities between the two encoding schemes exist. The work was a considerable speed-up compared to hand-coding the words from scratch, but manually disambiguating between lexeme candidates is still tedious work, and it motivated the current research to find additional methods for speeding up the encoding task.

In this article, we propose and evaluate new methods for *ranking lexeme suggestions for a word form* of a new word by generating lexeme candidates, i.e., base form and paradigm combinations, with an entry generator and then further generating key word

[1] http://joukahainen.puimula.org/

forms[2] for the lexeme candidates in order to *find support for the lexemes in a corpus* to weed out irrelevant lexeme suggestions. In section 2, we outline the directly related previous work. In section 3, we describe the new methods. In section 4, we present the training and test data. In section 5, we evaluate the model. In section 6, we discuss the method and the test results in light of the existing literature and some similar methods.

2 Lexicon-Based Entry Generator

To create entries for a morphological analyzer from previously unseen words and word forms, we need an entry generator. Ideally, we can use information that is already available in some existing morphological description to encode new entries in a similar fashion. Below, we briefly outline a general method for creating lexicon-based entry generators that was introduced by Lindén [11]. In his article, Lindén demonstrates that the method works well for English, Finnish and Swedish.

Assume that we have a finite-state transducer lexicon T which relates base forms, $b(w)$, to word forms, w . Let w belong to the input language L_I and $b(w)$ to the output language L_O of the transducer lexicon T . Our goal is to create an entry generator for word forms that are unknown to the lexicon, i.e., we wish to provide the most likely base forms $b(u)$ for an unknown input word u. In order to create an entry generator, we first define the left quotient and the weighted universal language with regard to a lexical transducer. For a general introduction to automata theory and weighted transducers, see e.g., [13].

We can regard the left quotient as the set of postfixes of L_1 that complete words from L_2 such that the resulting word is in L_1 . If L_1 and L_2 are formal languages, the left quotient of L_1 with regard to L_2 is the language consisting of strings w such that xw is in L_1 for some string x in L_2 . Formally, we write the left quotient as in (1).

$$L_1 \backslash L_2 = \{a | \exists x((x \in L_2)(xa \in L_1))\} \tag{1}$$

If L is a formal language with alphabet Σ, a universal language, U, is a language consisting of strings in Σ^*. The weighted universal language, W, is a language consisting of strings in Σ^* with weights $p(w)$ assigned to each string. For our purposes, we define the weight $p(w)$ to be proportional to the length of w. We define a weighted universal language as in (2).

$$W = \{w | \exists w(w \in \Sigma^*)\} \tag{2}$$

with weights $p(w) = C|w|$, where C is a constant.

A finite-state transducer lexicon, T, is a formal language relating the input language L_I to the output language L_O. The pair alphabet of T is the set of input and output symbol pairs related by T. An identity pair relates a symbol to itself.

[2] In highly inflecting languages like Finnish, it is not feasible to generate all word forms of a paradigm, as a noun can have more than 2000 word forms and a verb more than 10000 forms. A paradigm can be identified by a small set of inflected forms. This strategy is often used in lexicons intended for language learners to identify or illustrate verb paradigms for irregular verbs, e.g., in Romance or Germanic languages.

We create an entry generator, G, for the lexicon T by constructing the weighted universal language W for identity pairs based on the alphabet of L_1 concatenating it with the left quotient of T with regard to the universal language U of the pair alphabet of T as shown in (3).

$$G(T) = WT\backslash U \tag{3}$$

Lindén [11] proves that it is always possible to create an entry generator, $G(T) = WT\backslash U$, from a weighted lexical transducer T.

The model is general and requires no information in addition to the weighted lexicon from which the entry generator is derived. Therefore Lindén suggests that it be used as a baseline for other entry generator methods. For a sample output from the entry generator, see table 1.

3 Corpus-Based Lexeme Ranking

Assume that we have a morphological entry generator that generates a set of base form and paradigm combinations for out-of-vocabulary word forms. Each base form and paradigm combination defines a lexeme. In order to automatically score the lexemes suggested by the entry generator, we generate key word forms of the lexemes and look for the word forms in a corpus. Generally a lexeme whose key word forms are well-attested, i.e., many forms are in use and each form is used repeatedly, is more likely to be correct than a lexeme whose key word forms cannot be found or have only a few documented instances. Rare forms may even be spelling errors. By scoring all the lexemes provided by the entry generator, we can order the lexemes in descending order of support.

We have an unknown word form, w, for which we generate a set of lexeme candidates $U = \{l_1, l_2, l_3, \ldots l_n\}$. Each lexeme candidate, l_i, is defined by its set of word forms, from which we choose a set of key word forms, $K_i = \{k_1, k_2, k_3, \ldots k_m\}$, for scoring support of the lexeme.

We define a method for scoring the possible lexemes of an unknown word form by defining the probability, $P(l_i|w)$, of a lexeme, l_i, when given an unknown word form, w. Since we cannot directly estimate the conditional probability of a lexeme with regard to a word form from a corpus of running text, we use Bayes' rule as in (4) to reformulate the conditional probability.

$$P(l_i|w) = \frac{P(l_i, w)}{P(w)} = \frac{P(w|l_i) * P(l_i)}{P(w)} \tag{4}$$

The most likely lexeme l is provided by (4) by finding the l_i which maximizes the equation, in which $P(w)$ can be regarded as a constant. We then get (5) for l.

$$l = \arg\max_{l_i} P(w|l_i) * P(l_i) \tag{5}$$

We have $P(w|l_i)$ which is the probability of the original word form for a lexeme candidate, i.e., the probability that w is an inflected form of a lexeme candidate l_i, and the probability $P(l_i)$ of the lexeme in a large corpus. As a lexeme is defined by its set

of word forms, the probability of a lexeme in a corpus is the sum of the probabilities of its word forms in the corpus. We simplify the equation by assuming that the key word forms of a lexeme are sufficient to estimate the probability of the lexeme with the remaining word forms contributing a negligible constant addition, i.e., for a highly inflecting language, the key word forms should be chosen so that they represent a significant portion of the probability mass of the word forms of the lexeme. We sum over the key word forms and get (6),

$$l = \arg\max_{l_i} \sum_{k \in K_i} P(w|l_i) * P(k, l_i), \tag{6}$$

where $P(k, l_i)$ is the probability that a word form k belongs to lexeme l_i. To further simplify the equation, we assume that the conditional probability of the original word form in a suggested lexeme $P(w|l_i)$ is constant for our purposes, as no lexemes are suggested in which w could not appear as some inflected form, even if it may not be among the key word forms[3]. As a consequence of the assumption, the most likely lexeme l only depends on the innermost term of our equation, which further simplifies to (7).

$$l = \arg\max_{l_i} \sum_{k \in K_i} P(k, l_i), \tag{7}$$

To find the most likely lexeme, l, it is necessary to estimate the joint probability $P(k, l_i)$ that a key word form k co-occurs with lexeme l_i in a corpus.

3.1 Estimating Lexeme Likelihoods

In order to determine the likelihood that a word form co-occurs with a lexeme, we will look at three different methods for estimating this likelihood from a corpus. All three methods essentially regard the lexemes as small documents and the intention is to rank the documents, i.e., the lexemes, by their support in the corpus.

Key Word Indicator. The most basic method assumes that all key word forms are equally likely to appear in a lexeme. We define an indicator, $I(k)$, which is 1 or 0 depending on whether the word form k appears in the corpus or not. We call the basic method the *key word indicator scoring* and defined it in (8).

$$P(k, l_i) = \frac{I(k)}{|K_i|}, \tag{8}$$

[3] The even distribution of word forms in a lexeme is an oversimplification seemingly contradicting our previous assumption about key word forms representing the core probability mass. For consistency, we should have exploited the fact that a word form w, which is not among the key word forms, is relatively infrequent in the suggested lexeme by giving the lexeme a lower probability. In practice, we could have taken it into account, e.g., by filtering out lexeme suggestions that did not contain the original word form w among the key word forms effectively giving such lexeme suggestions 0 probability.

where $|K_i|$ is the number of key word forms that we investigate[4] for the lexeme l_i. If we always considered the same number of word forms for each lexeme, $|K_i|$ could be ignored. For some languages, it may be possible to look at all the word forms of a lexeme, but for some highly inflecting languages it is practical to use only a few key word forms for each lexeme. The number of key word forms may depend on the paradigm of the lexeme or even on the competing lexeme candidates in which case $|K_i|$ is needed as a normalizing factor.

Key Word Frequency. In order to better take into account the fact that a frequent word form is more significant for a lexeme than an infrequent one, which may even be a spelling error, we also consider the frequency, $F(k)$, of a word form k. However, it is unlikely that the importance of a word form is directly proportional to the frequency, so we consider the logarithm of the frequency. In addition, we need to smooth the frequency function by adding one, which creates a frequency scoring that mimics the indicator function for zero frequency key words, where $log(1) = 0$ and grows monotonically for larger frequencies. We thereby get a scoring method defined in (9) that is a variation of the term frequency for documents in information retrieval. We call this method the *key word frequency scoring*.

$$P(k,l_i) = \frac{\log(F(k))}{|K_i| * C}, \tag{9}$$

where C is a normalizing constant proportional to the logarithm of the number of tokens in the corpus. The constant has no effect on the ranking, but it serves to normalize the scoring into a probability distribution.

Key Word Frequency with Inverse Lexeme Frequency. A word form k may simultaneously belong to the set of word forms K_i of several candidate lexemes l_i. To take into account the distinctiveness of each key word form, we calculate a score similar to the inverse document frequency in information retrieval. The *inverse lexeme frequency*, $ilf(k)$, is equal to the logarithm of the number of lexeme candidates, n, divided by the number of candidates $|k \in K_i|$ in which k is a key word form. The distinctiveness score or the *inverse lexeme frequency* of a key word form is defined in (10).

$$ilf(k) = \log \frac{n}{|k \in K_i|}, \tag{10}$$

The key word frequency method is scaled with the distinctiveness score to yield (10). We call this method the *key word frequency with inverse lexeme frequency scoring*.

$$P(k,l_i) = \frac{\log(F(k))}{|K_i| * C} * ilf(k), \tag{11}$$

[4] The connection with the joint probability $P(k,l_i)$ is not obvious in the formula, however, the same key word form may affect two lexemes differently, e.g., the English word form *works* can be seen as one out of four forms of a verb but as one out of two forms of a noun contributing a different probability mass to each of the lexemes.

The scoring methods can be used with any data that reflects the occurrences of key words. Although we refer to the source of word frequency data as a corpus, the method can be used with other data sources as well. As is described in section 5, we have successfully tested the methods using both corpus material and page frequencies returned by a web search engine. In theory, the scoring methods should work with any data source that reflects the occurrence of words in language use.

4 Training and Test Data

To test our methods for corpus-based ranking of lexemes generated by a lexical entry generator, we use the entry generator for Finnish created by Lindén [11] and implemented with the Helsinki Finite-State Technology tools [14]. In 4.1, we briefly describe the lexical resources used for the finite-state transducer lexicon, which was converted into an entry generator.

Words unknown to the lexicon were drawn from a language-specific text collection. The correct entries for a sample of the unknown words were manually determined. In 4.2, we describe the text collections and, in 4.3, the samples used as test data. In 4.4, we describe the evaluation method and characterize the baseline.

4.1 Lexical Data for the Transducer Lexicon and Entry Generator

Lexical descriptions relate look-up words to other words and indicate the relation between them. A morphological finite-state transducer lexicon relates a word in dictionary form to all of its inflected forms. For an introduction, see e.g., [15].

Our current Finnish morphological analyzer was created by [16] based on the Finnish word list *Kotimaisten kielten tutkimuskeskuksen nykysuomen sanalista* [12], which contains 94110 words in base form. Of these, approximately 43000 are non-compound base forms with paradigm information. The word list consists of words in citation form annotated with a paradigm and possibly a gradation pattern. There are 78 paradigms and 13 gradation patterns. For example, the entry for *käsi* 'hand' is *käsi 27* referring to paradigm 27 without gradation, whereas the word *pato* 'dam' is given as *pato 1F* indicating paradigm 1 with gradation pattern F. From this description, a lexical transducer is compiled with a cascade of finite-state operations. For nominal paradigms, i.e., nouns and adjectives, inflection includes case inflection, possessive suffixes and clitics creating more than 2000 word forms for each nominal. For the verbal inflection, all tenses, moods and personal forms are counted as inflections, as well as all infinitives and participles and their corresponding nominal forms creating more than 10000 forms for each verb. In addition, the Finnish lexical transducer also covers nominal compounding.

This finite-state transducer lexicon was converted into an entry generator using the procedure outlined in section 2.

4.2 Data Collections for Word Counts

To test the general applicability of our scoring methods, we decided to use two different data sources. The first data source is a large text data collection of Finnish and the

second data source is the generally available search engine Google restricted to Finnish documents, which represents an even larger text collection.

The first data source is the *Finnish Text Collection*, which is an electronic document collection of the Finnish language. It consists of 180 million running text tokens. The corpus contains news texts from several current Finnish newspapers. It also contains extracts from a number of books containing prose text, including fiction, education and sciences. Gatherers are the Department of General Linguistics, University of Helsinki; The University of Joensuu; and CSC Scientific Computing Ltd. The corpus is available through CSC [www.csc.fi]. We used this text collection to provide frequency counts of word forms.

The second data source, i.e., Google on Finnish documents currently[5] indexes approximately 152 million documents, which provided the document counts, i.e., they are not direct word frequency counts, but the word frequency is of course reflected in the number of documents that the word appears in.

4.3 Test Data Collections

To test how well the scoring methods are able to rank the best lexeme among the top lexeme candidates for a new and previously unseen word, we used two different test word collections, for which the correct base form and paradigm combinations had been determined manually.

To test the methods, we used the test data collection developed by Lindén [11], which is a set of word forms drawn from the *Finnish Text Collection*. In order to extract word forms that represent relatively infrequent and previously unseen words, 5000 word and base form pairs had been drawn at random from the frequency rank 100001–300000. To get new words, only word forms that were not recognized by the lexical transducer were kept. However, from this test data, strings containing numbers, punctuation characters, or only upper case characters were also removed, as such strings require other forms of preprocessing in addition to some limited morphological analysis. Of the randomly selected strings, 1715 represented words not previously seen by the lexical transducer. For these strings, correct entries were created manually. Of these, only 48 strings had a verb form reading. The rest were noun or adjective readings. Only 43 had more than one possible reading.

A sample of the word forms from the first data set are: *ulkoasultaan* 'of its appearance', *kilpailulainsäädännön* 'legal framework on competition', *epätasa-arvoa* 'inequality', *euromaan* 'of a country using the euro', *työvoimapolitiikka* 'labor policy', *pariskunnasta* 'according to the married couple', *vastalausemyrskyn* 'of the objection storm', *liioitellun* 'of the exaggerated', *ruuanlaiton* 'of the cookery', *valtaannousun* 'of the ascent to power', *suurtapahtumaan*, *ostamiaan* 'the ones that they bought', ...

In table 1, we see an example of the word form *ulkoasultaan* and the suggested base forms and paradigms as they have been generated by the entry generator and expanded with key word forms in order for a scoring method to determine the best lexeme for a morphological entry.

Using the entry generator developed by Lindén [11], we developed a larger second test data collection based on the words of the Finnish open source dictionary project

[5] February, 2009 by searching for *ja* 'and' in Finnish documents.

Joukahainen[6]. We selected the words in the Joukahainen word list that were not included in the lexical data for the Finnish entry generator. Based on the existing lexical encoding of the Joukahainen project and the entry generator, the new words were encoded according to the guidelines of *The Research Institute for the Languages of Finland* [12] used in the entry generator. A list of 11026 new entries was created as test data. As the data came from an open-source word-list project, the words were all in base form.

Table 1. Word form *ulkoasultaan* 'of its appearance' and the key word forms of the lexemes suggested by the entry generator. The English glosses are added in the table for readability.

ulkoasu	*1 noun*	'appearance'
		ulkoasu ulkoasun ulkoasua ulkoasuun ulkoasut
		ulkoasujen ulkoasuja ulkoasuihin
ulkoasu	*2 noun*	'appearance'
		ulkoasu ulkoasun ulkoasua ulkoasuun ulkoasut
		ulkoasujen∼ulkoasuitten∼ulkoasuiden
		ulkoasuja∼ulkoasuita ulkoasuihin
ulkoasullata	*73 I verb*	'to stuff from the outside'
		ulkoasullata ulkoasultaan ulkoasultasi ulkoasultaisi
		ulkoasullannee ulkoasullatkoon ulkoasullannut ulkoasullattiin
ulkoasu	*21 noun*	'appearance'
		ulkoasu ulkoasun ulkoasuta ulkoasuhun ulkoasut
		ulkoasuiden ulkoasuita ulkoasuihin

4.4 Evaluation Measures, Baselines and Significance Test

We report our test results using recall and average precision at maximum recall. *Recall* means all the word forms in the test data for which an accurate base form suggestion is produced. *Average precision at maximum recall* is an indicator of the amount of noise that precedes the intended paradigm suggestions, where n incorrect suggestions before the m correct ones give a precision of $1/(n+m)$, i.e., no noise before a single intended base form per word form gives 100% precision on average, and no correct suggestion at maximum recall gives 0% precision. The *F-score* is the harmonic mean of the recall and the average precision. We will use only the recall and average precision among the top 6 candidates, as the output is intended for human post processing. In general, this will give us a lower, i.e., more conservative, recall than considering all candidates.

The random baseline for Finnish is that the correct entry is one out of 78 paradigms with one out of 13 gradations, i.e., a random correct guess would on the average end up as guess number 507.

As suggested by Lindén [11], we use the automatically derived entry generator from section 4.1 as a baseline. Using his test data, the test results will be directly comparable to the baseline provided in table 2 with recall 82%, average precision 76% and the F-score 79%.

[6] http://joukahainen.puimula.org/

Table 2. Baseline for Finnish entry generator on infrequent word forms

Rank	Frequency	Percentage
#1	1140	66.5 %
#2	186	10.8 %
#3	64	3.7 %
#4	17	1.0 %
#5	4	0.2 %
#6	2	0.1 %
#7-∞	302	17.6 %
Total	1715	100.0 %

We also ran the entry generator directly on the base forms of our test data from the Joukahainen word collection in order to get the baseline provided in table 3 indicating 66% average precision and 72% recall with an F-score of 69%.

Table 3. Baseline for Finnish entry generator on list of new base forms

Rank	Frequency	Percentage
#1	6043	54.8 %
#2	1196	10.8 %
#3	482	4.4 %
#4	157	1.4 %
#5	64	0.6 %
#6	11	0.1 %
#7-∞	3073	27.9 %
Total	11026	100.0 %

The significance of the difference between the baselines and the tested scoring methods is determined with matched pairs. The Wilcoxon Matched-Pairs Signed-Ranks Test indicates whether the changes in the ranking differences are statistically significant. For large numbers the test is almost as sensitive as the Matched-Pairs Student t-test even if it does not assume a normal distribution of the ranking differences.

5 Evaluation

We test how well the lexeme scoring methods outlined in section 3 are able to select the best lexemes for a new word form using the test data described in section 4.2. Word forms representing previously unseen words were used as test data in the experiment. The generated entries are intended for human post-processing, so the first correct entry suggestion should be among the top 6 candidates, otherwise the ranking is considered a failure.

If a ranking method ranks several candidates with the same score, a stable sorting algorithm will keep the original order of the lexemes. To test the effect of the proposed

corpus-based ranking methods independently from the ordering given by the entry generator, the order of the entries generated from the entry generator were randomized before they were submitted to the corpus-method so as not to bias the corpus-method with the ranking order of the entry generator. We did five randomized runs for each evaluation and took the average of the ranking results.

In 5.1, we test the lexeme scoring with counts from the Finnish text corpus on the set of infrequent word forms. In 5.2, we test the scoring methods using page counts from the internet based on the Google search engine for the same set of word forms. In 5.3, we test the scoring methods on the set of new base forms using counts from the Finnish text corpus.

5.1 Corpus-Based Lexeme Ranking of Word Forms

We evaluate the lexeme ranking method on base forms and paradigms generated by the lexicon-based entry generator from the test set of infrequent word forms using word counts from the *Finnish Text Collection* described in section 4.2. In table 4, we see the precision, recall and F-score of the three scoring methods.

Table 4. Precision, Recall and F-score of scoring methods

Method	Precision	Recall	F-Score
key word indicator	79 %	89 %	84 %
key word frequency	72 %	84 %	78 %
key word frequency with ilf	76 %	87 %	81 %

The *key word indicator scoring* performed best and ranked a correct entry among the top 6 candidates for 89% of the test data as shown in table 5, which corresponds to an average position of 1.96 for the first correct entry with 89% recall and 79% average precision, i.e., an 84% F-score. All methods were statistically very highly significantly different using the corpus count data.

Table 5. Ranks of all the first correct lexeme suggestions using a text collection

Rank	Frequency	Percentage
#1	1184	69.0%
#2	229	13.4%
#3	71	4.1%
#4	24	1.4%
#5	15	0.9%
#6	7	0.4%
#7-∞	185	10.8%
Total	1715	100.0%

5.2 Page Count-Based Lexeme Ranking of Word Forms

We also evaluate the lexeme ranking method on base forms and paradigms generated by the lexicon-based entry generator from the test set of infrequent word forms using the World-Wide Web page counts for pages retrieved over a period of some weeks from Google for key words of the paradigms. We retrieved the data from pages which Google gave a Finnish language code. We used this as a way to verify the ranking methods on an independent data collection. In table 6, we see the precision, recall and F-score of the three scoring methods.

Table 6. Precision, Recall and F-score of scoring methods

Method	Precision	Recall	F-Score
key word indicator	68%	88%	77%
key word frequency	73%	83%	78%
key word frequency with ilf	74%	83%	78%

The *key word frequency scoring with inverse lexeme frequency* had the best overall performance and ranked a correct entry among the top 6 candidates for 83% of the test data as shown in table 7, which corresponds to an average position of 2.4 for the first correct entry with 83% recall and 74% average precision, i.e., an 78% F-score. The difference to the pure frequency method was not statistically significant. However, the *key word indicator scoring* had the best recall of 88%. The difference to the winning method was statistically significant on the lowest significance level.

Table 7. Ranks of all the first correct lexemes using page counts from the World-Wide Web

Rank	Frequency	Percentage
#1	1114	65.0%
#2	139	8.1%
#3	63	3.7%
#4	58	3.4%
#5	32	1.9%
#6	20	1.2%
#7-∞	289	16.9%
Total	1715	100.0%

5.3 Corpus-Based Lexeme Ranking of Base Forms

We evaluate the lexeme ranking method on base forms and paradigms generated by the lexicon-based entry generator for base forms in our subset of the Joukahainen word list using counts from the *Finnish Text Collection* described in section 4.2. In table 8, we see the precision, recall and F-score of the three scoring methods.

Table 8. Precision, Recall and F-score of scoring methods

Method	Precision	Recall	F-Score
key word indicator	56%	78%	65%
key word frequency	50%	71%	59%
key word frequency with ilf	51%	72%	59%

The *key word indicator scoring* performed best and ranked a correct entry among the top 6 candidates for 78% of the test data as shown in table 9, which corresponds to an average position of 3.1 for the first correct entry with 78% recall and 56% average precision, i.e., an 65% F-score. The indicator scoring was highly statistically significantly better than the two other scoring methods, which had no statistical difference between them.

Table 9. Ranks of all the first lexemes using counts from the text collection

Rank	Frequency	Percentage
#1	4198	38.0%
#2	2038	18.5%
#3	1018	9.2%
#4	687	6.2%
#5	381	3.5%
#6	296	2.7%
#7-∞	2418	21.9%
Total	11026	100.0%

5.4 Significance

All the ranking methods of the lexemes from the morphological entry generator were statistically highly significantly better than the random baselines according to the Wilcoxon Matched-Pairs Signed-Ranks Test. The difference between the two winning methods, i.e., the *key word indicator scoring* and the *key word frequency scoring with inverse lexeme frequency*, were statistically significant on all the test data collections.

The improvement in the recall of 6–7% percentage points from the baseline model on two separate test data collections using the corpus word counts is also significant in practice as we counted words below the 6^{th} position as out-of-reach. This means that a significant number of additional correct classifications were now visible to the native speaker doing the entry revising.

6 Discussion

In this section, we discuss the results and give a brief overview of some related work. In 6.1, we compare our test results with previous efforts. In 6.2, we discuss some future work.

6.1 Discussion of Results

The problem when dealing with relatively low-frequency words is that an approach based on generating additional word forms of the lexemes may not contribute much. It may well be that the word form we are trying to classify is the only instance of the lexeme in the corpus. In that sense, turning to the internet for help seems like a good idea. It turned out that three times as many[7] of the generated word forms could be found on the internet as we were able to find in the 182 million word data collection of Finnish words.

It is interesting to note how the smaller amount of data affected the preference for the key word indicator scoring function. One reason for the good performance of the word forms using the text collection is of course that the new words and the text collection were ideally matched, when the new words were drawn from the same collection as the counts. However, this is not a severe limitation. New words are usually drawn from known text collections. That is often one of the purposes for collecting the texts in the first place. However, also when the lexemes from an independent data collection are ranked with the same kind of word counts, it turns out that the key word indicator scoring is the most successful. As can be seen, the precision drops below that of the baseline suggesting that the additional lexemes that become visible to the lexicographer do not make it all the way to the top. Currently, we believe that this is due to the smaller size of the corpus and the fact that it may be slightly cleaner. In order for the frequency scoring to become more effective than the indicator scoring, we need a sufficiently large number of word forms for which a word count is available.

Using the Google data, the frequency ranking becomes more effective. The explanation for this seems to be that a larger number of word forms for each paradigm slightly reduces the distinctions between some of the best lexeme suggestions. This makes it interesting to use the more fine-grained frequency scoring which can be further scaled to emphasize the distinctiveness of the key word forms.

Sometimes a misspelling may have been more common than a correctly spelled word. For example, the sixth highest scoring word in our material was *seuraavä*, with approx. 21000000 page counts, while its correctly spelled form, *seuraava*, had almost 500000 page counts less. This was in most cases corrected by a higher average frequency of the remaining key word forms in the correct lexeme. Sometimes the incorrect lexemes happened to contain a homonym of some frequently occurring word, which raised the score of the lexeme above that of the correct lexeme candidate.

The fact that as a source for ranking entries, the corpus data fared slightly better than the internet may in our case also be attributable to the fact that Finnish word forms in the frequency range 100000–300000 may not be so rare after all due to the rich morphology and productive compounding mechanism of Finnish.

The larger test data collection was definitely less suited to the text collection. It also had a lower baseline to begin with indicating that base forms may not be the ideal words to classify with an entry generator that has been created for generating entries for any inflection. However, when constructing the test data we could benefit from the base forms by automatically discarding lexeme suggestions that did not have the input base form among the key word suggestions, e.g., if we know that we are looking for an

[7] Data retrieved in December, 2008.

entry for the English base form *swimming*, good lexeme candidates like *swim V* can be mechanically discarded in favor of *swimming N*. This option was not used when testing the performance of the ranking methods, as we wanted to evaluate their contributions independently.

It remains to be seen how the base form data collection would perform using Google page counts. Would the higher frequency counts for more key word forms favor a more sensitive scoring method in the same way as it did on the smaller test data collection? As the methods were statistically significantly different on the smaller data set, we have reason to believe that this prediction will hold.

From a practical point of view, we were able to significantly reduce the workload of encoding lexical entries as much of the task can now be accomplished automatically by suggesting only those lexemes for which there is at least some support in the corpus in addition to the word form being investigated. However, a significant practical change is that assigning base forms and paradigms to words which previously required an expert lexicographer can now be accomplished by a native speaker making a choice between a very limited set of lexeme suggestions.[8]

6.2 Comparison with Similar or Related Efforts

A related idea of expanding key word forms of paradigms to identify new words and their paradigms has been suggested by Forsberg et al. [17]. However, their approach was to automatically deduce rules for which they could find as much support as was logically possible in order to make a safe inference. This leads to safely extracting words that already have a number of word forms in the corpus, i.e., mid or high-frequency words, which for all practical purposes have most likely already been encoded and are readily available in public domain morphological descriptions like the Ispell dictionaries [9] or more advanced descriptions like the Finnish dictionary *Kotimaisten kielten tutkimuskeskuksen nykysuomen sanalista* [12]. It should be noted that Forsberg et al. [17] drew the conclusion that it is recommendable that a linguist writes the extraction rules. In addition, they used an even smaller corpus than we did. This indicates that their reliance on an approach that is most similar to our *key word indicator scoring* will probably be less effective when the corpus-size increases, requiring their linguist to write additional constraints for the extraction rules.

The approach suggested by Mikheev [1,2] aims at solving the issue of unknown words in the context of part-of-speech taggers. However, in that context the problem is slightly simpler as the guesser only needs to identify a likely part of speech, not the base form and the full inflectional paradigm of a word. He suggests an automatic way of extracting prefix and postfix patterns for guessing the part of speech. A related approach aiming at inducing paradigms for words and inflectional morphologies for 30 different languages is suggested by [4].

Since there is a growing body of translated text even for less studied languages, there are interesting approaches using multi-lingual evidence for inducing morphologies, see, e.g., Yarowsky and Wicentowski [18]. This approach is particularly fruitful if we can benefit from the similarities of closely related languages.

[8] For a demo of the classification interface, see http://www.ling.helsinki.fi/cgi-bin/omor/omorfi-cgidemo.py

If we cannot find enough support for any particular lexeme for a given word form, e.g., if the word is too infrequent so that there are no other inflections than the original word form, we need a way to make inferences based on related or similar strings. We need to make inferences based on the analogy with already known words as suggested, e.g., by Goldsmith [5] or Lindén [8,11,19].

6.3 Future Work

The current study aimed at evaluating the effect of corpus evidence in isolation. We have reason to believe that an approach which combines the output of the entry generator with the methods evaluated in this article, e.g., by relying on the ordering suggested by the entry generator when the corpus evidence does not distinguish between the ranks of the lexeme suggestions is an effective way of combining the corpus evidence with the entry generator. This essentially means that the entry generator functions as fall-back when there is a lack of corpus evidence.

Currently we only extract inflectional information in the form of lexical entries, even if the context of a new word also contributes other types of lexical information such as part of speech, argument structure and other more advanced types of syntactic and semantic information.

It is important to note that our experiment verifies that the lexeme ranking can be performed using page counts instead of word counts with a sufficiently large document corpus, which is by no means self-evident. Many of the word forms will refer to the same pages, which also opens up avenues for future research. One could perhaps use page counts for a combination of the base form with some other word form of a lexeme in order to reduce the noise by searching directly for pages with combinations of several key word forms.

The Internet in addition to page counts also provides some context for a word form. Essentially this means that we have access to the local semantic context of a word. The Internet is an ever-changing medium and any linguistic data derived form it is subject to change. The challenge is to harness this evidence to distill the information inherent in large numbers while still adapting to the significant changes in language use.

7 Conclusions

We have proposed and successfully tested new methods for ranking lexemes generated for word forms of new words using additional corpus evidence for key word forms of the lexemes suggested by an entry generator. We tested the model on Finnish, which is a highly inflecting language with a considerable set of inflectional paradigms and stem change categories. A key finding was that the ranking functions that can better take into account fine-grained words counts seems likely to perform better on larger and perhaps inevitably more noisy corpora. We tested the effects of the methods independently from the prior ranking by the entry generator. The methods can be combined to provide optimal performance.

Our corpus-based ranking methods have 56–79% average precision and 78–89% recall among the top 6 candidates, i.e., an F-score of 65–84%, indicating that the first

correct entry suggestion is on the average found in positions 1.9–3.1. The ranking methods based on corpus evidence were found to be significant in practice by increasing the recall among the top 6 candidates with 7–8%, which saves the lexicographer some work when reducing the need to create entries from scratch.

Acknowledgments

We are grateful to the Finnish Academy. We are also grateful to Inari Listenmaa, who used her native language skills to evaluate the entry generator by revising entries for new words from the Joukahainen word list.

References

1. Mikheev, A.: Unsupervised Learning of Word Category Guessing Rules. In: Proc. of the 34th Annual Meeting of the Association for Computational Linguistics (ACL 1996), pp. 327–334 (1996)
2. Mikheev, A.: Automatic Rule Induction for Unknown-Word Guessing. Computational Linguistics 23(3), 405–423 (1997)
3. Oflazer, K., Nirenburg, S., McShane, M.: Bootstrapping Morphological Analyzers by Combining Human Elicitation and Machine Learning. Computational Linguistics 27(1), 59–85 (2001)
4. Wicentowski, R.: Modeling and Learning Multilingual Inflectional Morphology in a Minimally Supervised Framework. PhD Thesis. John Hopkins University, Baltimore, USA (2002)
5. Goldsmith, J.A.: Morphological Analogy: Only a Beginning (2007), http://hum.uchicago.edu/~jagoldsm/Papers/analogy.pdf
6. Creutz, M., Hirsimäki, T., Kurimo, M., Puurula, A., Pylkkönen, J., Siivola, V., Varjokallio, M., Arisoy, E., Saraçlar, M., Stolcke, A.: Morph-based speech recognition and modeling of out-of vocabulary words across languages. ACM Transactions on Speech and Language Processing 5(1), article 3 (2007)
7. Kurimo, M., Creutz, M., Turunen, V.: Overview of Morpho Challenge in CLEF 2007. In: Working Notes of the CLEF 2007 Workshop, pp. 19–21 (2007)
8. Lindén, K.: A Probabilistic Model for Guessing Base Forms of New Words by Analogy. In: Gelbukh, A. (ed.) CICLing 2008. LNCS, vol. 4919, pp. 106–116. Springer, Heidelberg (2008)
9. Kuenning, G.: Dictionaries for International Ispell (2007), http://www.lasr.cs.ucla.edu/geoff/ispelldictionaries.html
10. Lingsoft, Inc.: Demos, http://www.lingsoft.fi/?doc_id=107&lang=en
11. Lindén, K.: Guessers for Finite-State Transducer Lexicons. In: CICling-2009, 10th International Conference on Intelligent Text Processing and Computational Linguistics, Mexico City, Mexico, March 1- 7 (2009)
12. Kotimaisten kielten tutkimuskeskuksen nykysuomen sanalista. Research Institute for the Languages of Finland (2007), http://kaino.kotus.fi/sanat/nykysuomi/
13. Sakarovitch, J.: Éléments de théorie des automates. Vuibert, Paris (2003)
14. HFST: Helsinki Finite-State Technology (2008), http://www.ling.helsinki.fi/kieliteknologia/tutkimus/hfst/index.shtml
15. Koskenniemi, K.: Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production. PhD Thesis. Department of General Linguistics, University of Helsinki, Publication No. 11 (1983)

16. Pirinen, T.: Open Source Morphology for Finnish using Finite-State Methods (in Finnish). Technical Report. Department of Linguistics, University of Helsinki (2008)
17. Forsberg, M., Hammarström, H., Ranta, A.: Morphological Lexicon Extraction from Raw Text Data. In: Salakoski, T., Ginter, F., Pyysalo, S., Pahikkala, T. (eds.) FinTAL 2006. LNCS (LNAI), vol. 4139, pp. 488–499. Springer, Heidelberg (2006)
18. Yarowsky, D., Wicentowski, R.: Minimally Supervised Morphological Analysis by Multimodal Alignment. In: Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics (2000)
19. Lindén, K.: Entry Generation by Analogy—Encoding New Words for Morphological Lexicons. Northern European Journal of Language Technology (May 2009)

Author Index