Hunmorph: open source word analysis

Viktor Trón
IGK, U of Edinburgh
2 Buccleuch Place
EH8 9LW Edinburgh
v.tron@ed.ac.uk

György Gyepesi
K-PRO Ltd.
H-2092 Budakeszi
Villám u. 6.
ggyepesi@kpro.hu

Péter Halácsy
Centre of Media Research and Education
Stoczek u. 2
H-1111 Budapest
hp@mokk.bme.hu

András Kornai
MetaCarta Inc.
350 Massachusetts Avenue
Cambridge MA 02139
andras@kornai.com

László Németh
CMRE
Stoczek u. 2
H-1111 Budapest
nemeth@mokk.bme.hu

Dániel Varga
CMRE
Stoczek u. 2
H-1111 Budapest
daniel@mokk.bme.hu

Abstract

Common tasks involving orthographic words include spellchecking, stemming, morphological analysis, and morphological synthesis. To enable significant reuse of the language-specific resources across all such tasks, we have extended the functionality of the open source spellchecker `MySpell`, yielding a generic word analysis library, the runtime layer of the `hunmorph` toolkit. We added an offline resource management component, `hunlex`, which complements the efficiency of our runtime layer with a high-level description language and a configurable precompiler.

0 Introduction

Word-level analysis and synthesis problems range from strict recognition and approximate matching to full morphological analysis and generation. Our technology is predicated on the observation that all of these problems are, when viewed algorithmically, very similar: the central problem is to dynamically analyze complex structures derived from some lexicon of base forms. Viewing word analysis routines as a unified problem means sharing the same codebase for a wider range of tasks, a design goal carried out by finding the parameters which optimize each of the analysis modes independently of the language-specific resources.

The C/C++ runtime layer of our toolkit, called `hunmorph`, was developed by extending the codebase of `MySpell`, a reimplementation of the well-known `Ispell` spellchecker. Our technology, like the `Ispell` family of spellcheckers it descends from, enforces a strict separation between the language-specific resources (known as `dictionary` and `affix` files), and the runtime environment, which is independent of the target natural language.

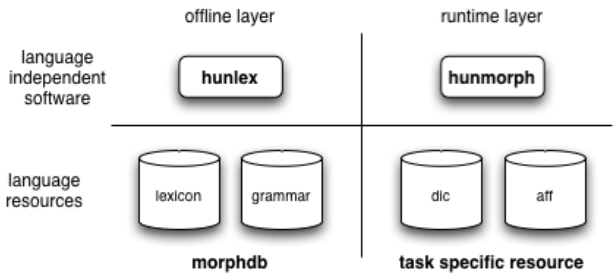

Figure 1: Architecture

Compiling accurate wide coverage machine-readable dictionaries and coding the morphology of a language can be an extremely labor-intensive task, so the benefit expected from reusing the language-specific input database across tasks can hardly be overestimated. To facilitate this resource sharing and to enable systematic task-dependent optimizations from a central lexical knowledge base, we designed and implemented a powerful offline layer we call `hunlex`. `Hunlex` offers an easy

to use general framework for describing the lexicon and morphology of any language. Using this description it can generate the language-specific `aff/dic` resources, optimized for the task at hand. The architecture of our toolkit is depicted in Figure 1. Our toolkit is released under a permissive LGPL-style license and can be freely downloaded from `mokk.bme.hu/resources/hunmorph`.

The rest of this paper is organized as follows. Section 1 is about the runtime layer of our toolkit. We discuss the algorithmic extensions and implementational enhancements in the C/C++ runtime layer over `MySpell`, and also describe the newly created Java port `jmorph`. Section 2 gives an overview of the offline layer `hunlex`. In Section 3 we consider the free open source software alternatives and offer our conclusions.

1 The runtime layer

Our development is a prime example of code reuse, which gives open source software development most of its power. Our codebase is a direct descendant of `MySpell`, a thread-safe C++ spell-checking library by Kevin Hendricks, which descends from `Ispell` Peterson (1980), which in turn goes back to Ralph Gorin's `spell` (1971), making it probably the oldest piece of linguistic software that is still in active use and development (see `fmg-www.cs.ucla.edu/fmg-members/geoff/ispell.html`).

The key operation supported by this codebase is *affix stripping*. Affix rules are specified in a static resource (the `aff` file) by a sequence of conditions, an append string, and a strip string: for example, in the rule forming the plural of *body* the strip string would be *y*, and the affix string would be *ies*. The rules are reverse applied to complex input wordforms: after the append string is stripped and the edge conditions are checked, a pseudo-stem is hypothesized by appending the strip string to the stem which is then looked up in the base dictionary (which is the other static resource, called the `dic` file).

Lexical entries (base forms) are all associated with sets of *affix flags*, and affix flags in turn are associated to sets of affix rules. If the hypothesized base is found in the dictionary after the reverse application of an affix rule, the algorithm checks whether its flags contain the one that the affix rule is assigned to. This is a straight table-driven approach, where affix flags can be interpreted directly as lexical features that license entire subparts of morphological paradigms. To pick applicable affix rules efficiently, `MySpell` uses a fast indexing technique to check affixation conditions.

In theory, affix-rules should only specify genuine prefixes and suffixes to be stripped before lexical lookup. But in practice, for languages with rich morphology, the affix stripping mechanism is (ab)used to strip complex clusters of affix morphs in a single step. For instance, in Hungarian, due to productive combinations of derivational and inflectional affixation, a single nominal base can yield up to a million word forms. To treat all these combinations as affix clusters, legacy ispell resources for Hungarian required so many combined affix rule entries that its resource file sizes were not manageable.

To solve this problem we extended the affix stripping technique to a multistep method: after stripping an affix cluster in step i, the resulting pseudo-stem can be stripped of affix clusters in step $i+1$. Restrictions of rule application are checked with the help of flags associated to affixes analogously to lexical entries: this only required a minor modification of the data structure coding affix entries and a recursive call for affix stripping. By cross-checking flags of prefixes on the suffix (as opposed to the stem only), simultaneous prefixation and suffixation can be made interdependent, extending the functionality to describe circumfixes like German participle *ge+t*, or Hungarian superlative *leg+bb*, and in general provide the correct handling of prefix-suffix dependencies like English *undrinkable* (cf. **undrink*), see Németh et al. (2004) for more details.

Due to productive compounding in a lot of languages, proper handling of composite bases is a feature indispensable for achieving wide coverage. `Ispell` incorporates the possibility of specifying lexical restrictions on compounding implemented as switches in the base dictionary. However, the algorithm allows any affixed form of the bases that has the relevant switch to be a potential member

of a compound, which proves not to be restrictive enough. We have improved on this by the introduction of position-sensitive compounding. This means that lexical features can specify whether a base or affix can occur as leftmost, rightmost or middle constituent in compounds and whether they can *only* appear in compounds. Since these features can also be specified on affixes, this provides a welcome solution to a number of residual problems hitherto problematic for open-source spellcheckers. In some Germanic languages, 'fogemorphemes', morphemes which serve linking compound constituents can now be handled easily by allowing position specific compound licensing on the foge-affixes. Another important example is the German common noun: although it is capitalized in isolation, lowercase variants should be accepted when the noun is a compound constituent. By handling lowercasing as a prefix with the compound flag enabled, this phenomenon can be handled in the resource file without resort to language specific knowledge hard-wired in the code-base.

1.1 From spellchecking to morphological analysis

We now turn to the extensions of the `MySpell` algorithm that were required to equip `hunmorph` with stemming and morphological analysis functionality. The core engine was extended with an optional output handling interface that can process arbitrary string tags associated with the affix-rules read from the resources. Once this is done, simply outputting the stem found at the stage of dictionary lookup already yields a stemmer. In multistep affix stripping, registering output information associated with the rules that apply renders the system capable of morphological analysis or other word annotation tasks. Thus the *processing of output tags* becomes a mode-dependent parameter that can be:

- switched off (spell-checking)
- turned on only for tag lookup in the dictionary (simple stemming)
- turned on fully to register tags with all rule-applications (morphological analysis)

The single most important algorithmic aspect that distinguishes the recognition task from analysis is the handling of ambiguous structures. In the original `MySpell` design, identical bases are conflated and once their switch-set licensing affixes are merged, there is no way to tell them apart. The correct handling of homonyms is crucial for morphological analysis, since base ambiguities can sometimes be resolved by the affixes. Interestingly, our improvement made it possible to rule out homonymous bases with incorrect simultaneous prefixing and suffixing such as English *out+number+'s*. Earlier these could be handled only by lexical pregeneration of relevant forms or duplication of affixes.

Most importantly, ambiguity arises in relation to the number of analyses output by the system. While with spell-checking the algorithm can terminate after the first analysis found, performing an exhaustive search for all alternative analyses is a reasonable requirement in morphological analysis mode as well as in some stemming tasks. Thus the *exploration of the search space* also becomes an active parameter in our enhanced implementation of the algorithm:

- search until the first correct analysis
- search restricted multiple analyses (e.g., disabling compounds)
- search all alternative analyses

Search until the first analysis is a functionality for recognizers used for spell-checking and stemming for accelerated document indexing. Preemption of potential compound analyses by existing lexical bases serves as a general way of filtering out spurious ambiguities when a reduction is required in the space of alternative analyses. In these cases, frequent compounds which trick the analyzer can be precompiled to the lexicon. Finally, there is a possibility to give back a full set of possible analyses. This output then can be passed to a tagger that disambiguates among the candidate analyses. Parameters can be used that guide the search (such as 'do lexical lookup first at all stages' or 'strip the shortest affix first'), which yield candidate rankings without the use of numerical weights or statis-

tics. These rankings can be used as disambiguation heuristics based on a general idea of blocking (e.g., *Times* would block an analysis of *time+s*). All further parametrization is managed offline by the resource compiler layer, see Section 2.

1.2 Reimplementing the runtime layer

In our efforts to gear up the `MySpell` codebase to a fully functional word analysis library we successfully identified various resource-related, algorithmic and implementational bottlenecks of the affix-rule based technology. With these lessons learned, a new project has been launched in order to provide an even more flexible and efficient open source runtime layer. A principled object-oriented refactorization of the same basic algorithm described above has already been implemented in Java. This port, called `jmorph` also uses the `aff/dic` resource formats.

In `jmorph`, various algorithmic options guiding the search (shortest/longest matching affix) can be controlled for each individual rule. The implementation keeps track of affix and compound matches checking conditions only once for a given substring and caching partial results. As a consequence, it ends up being measurably faster than the C++ implementation with the same resources.

The main loop of `jmorph` is driven by configuring *consumers*, i.e., objects which monitor the recursive step that is running. For example the analysis of the form *beszédesek* 'talkative.PLUR' begins by inspecting the global configuration of the analysis: this initial consumer specifies how many analyses, and what kind, need to be found. In Step 1, the initial consumer finds the rule that strips *ek* with stem *beszédes*, builds a consumer that can apply this rule to the output of the analysis returned by the next consumer, and launches the next step with this consumer and stem. In Step 2, this consumer finds the rule stripping *es* with stem *beszéd*, which is found in the lexicon. *beszéd* is not just a string, it is a complete lexical object which lists the rules that can apply to it and all the homonyms. The consumer creates a new analysis that reflects that *beszédes* is formed from *beszéd* by suffixing *es* (a suffix object), and passes this back to its parent consumer, which verifies whether the *ek* suffixation rule is applicable. If not, the Step 1 consumer requests further analyses from the Step 2 consumer. If, however, the answer is positive, the Step 1 consumer returns its analysis to the Step 0 (initial) consumer, which decides whether further analyses are needed.

In terms of functionality, there are a number of differences between the Java and the C++ variants. `jmorph` records the full parse tree of rule applications. By offering various ways of serializing this data structure, it allows for more structured information in the outputs than would be possible by simple concatenation of the tag chunks associated with the rules. Class-based restrictions on compounding is implemented and will eventually supersede the overgeneralizing position-based restrictions that the C++ variant and our resources currently use.

Two major additional features of `jmorph` are its capability of morphological synthesis as well as acting as a guesser (hypothesizing lemmas). Synthesis is implemented by forward application of affix rules starting with the base. Rules have to be indexed by their tag chunks for the search, so synthesis introduces the non-trivial problem of chunking the input tag string. This is currently implemented by plug-ins for individual tag systems, however, this should ideally be precompiled offline since the space of possible tags is limited.

2 Resource development and offline precompilation

Due to the backward compatibility of the runtime layer with `MySpell`-style resources, our software can be used as a spellchecker and simplistic stemmer for some 50 languages for which `MySpell` resources are available, see `lingucomponent.openoffice.org/spell_dic.html`.

For languages with complex morphology, compiling and maintaining these resources is a painful undertaking. Without using a unified framework for morphological description and a principled method of precompilation, resource developers for highly agglutinative languages like Hungarian (see `magyarispell.sourceforge.net`) have to resort to a maze of scripts to maintain and precompile `aff` and `dic` files. This problem is intolerably magnified once morphological tags or additional

lexicographic information are to be entered in order to provide resources for the analysis routines of our runtime layer.

The offline layer of our toolkit seeks to remedy this by offering a high-level description language in which grammar developers can specify rule-based morphologies and lexicons (somewhat in the spirit of `lexc` Beesley and Karttunen (2003), the frontend to Xerox's Finite State Toolkit). This promises rapid resource development which can then be used in various tasks. Once primary resources are created, `hunlex`, the offline precompiler can generate `aff` and `dic` resources optimized for the runtime layer based on various compile-time configurations.

Figure 2 illustrates the description language with a fragment of English morphology describing plural formation. Individual rules are separated by commas. The syntax of the rule descriptions organized around the notion of *information blocks*. Blocks are introduced by keywords (like `IF:`) and allow the encoding of various properties of a rule (or a lexical entry), among others specifying affixation (`+es`), substitution, character truncation before affixation (`CLIP: 1`), regular expression matches (`MATCH: [^o]o`), positive and negative lexical feature conditions on application (`IF: f-v_altern`), feature inheritance, output (continuation) references (`OUT: PL_POSS`), output tags (`TAG: "[PLUR]"`).

One can specify the rules that can be applied to the output of a rule and also one can specify application conditions on the input to the rule. These two possibilities allow for many different styles of morphological description: one based on input feature constraints, one based on continuation classes (paradigm indexes), and any combination between these two extremes. On top of this, regular expression matches on the input can also be used as conditions on rule application.

Affixation rules "grouped together" here under `PLUR` can be thought of as allomorphic rules of the plural morpheme. Practically, this allows information about the morpheme shared among variants (e.g., morphological tag, recursion level, some output information) to be abstracted in a *preamble* which then serves as a default for the individual rules. Most importantly, the grouping of rules

```
PL
        TAG: "[PLUR]"
        OUT: PL_POSS
# house -> houses
, +s   MATCH: [^shoxy]    IF: regular
# kiss -> kisses
, +es  MATCH: [^c]s       IF: regular
# ...
# ethics
, +    MATCH: cs          IF: regular
# body -> bodies <C> is a regexp macro
, +ies MATCH: <C>y CLIP:1 IF: regular
# zloty -> zlotys
, +s   MATCH: <C>y       IF: y-ys
# macro -> macros
, +s   MATCH: [^o]o      IF: regular
# potato -> potatoes
, +es  MATCH: [^o]o      IF: o-oes
# wife -> wives
, +ves MATCH: fe CLIP: 2 IF: f-ves
# leaf -> leaves
, +ves MATCH: f  CLIP: 1 IF: f-ves
;
```

Figure 2: `hunlex` grammar fragment

into morphemes serves to index those rules which can be referenced in output conditions, For example, in the above the plural morpheme specifies that the plural possessive rules can be applied to its output (`OUT: PL_POSS`). This design makes it possible to handle some morphosyntactic dimensions (part of speech) very cleanly separated from the conditions regulating the choice of allomorphs, since the latter can be taken care of by input feature checking and pattern matching conditions of rules. The lexicon has the same syntax as the grammar only that morphemes stand for lemmas and variant rules within the morpheme correspond to stem allomorphs.

Rules with zero affix morph can be used as *filters* that decorate their inputs with features based on their orthographic shape or other features present. This architecture enables one to let only exceptions specify certain features in the lexicon while regular words left unspecified are assigned a default feature by the filters (see `PL_FILTER` in

```
REGEXP: C [bcdfgklmnprstvwxyz];

DEFINE: N
  OUT: SG PL_FILTER
  TAG: NOUN
;

PL_FILTER
  OUT:
       PL
  FILTER:
       f-ves
       y-ys
       o-oes
       regular
, DEFAULT:
       regular
;
```

Figure 3: Macros and filters in `hunlex`

Figure 3) potentially conditioned the same way as any rule application. Feature inheritance is fully supported, that is, filters for particular dimensions of features (such as the plural filter in Figure 3) can be written as independent units. This design makes it possible to engineer sophisticated filter chains decorating lexical items with various features relevant for their morphological behavior. With this at hand, extending the lexicon with a regular lexeme just boils down to specifying its base and part of speech. On the other hand, independent sets of filter rules make feature assignments transparent and maintainable.

In order to support concise and maintainable grammars, the description language also allows (potentially recursive) macros to abbreviate arbitrary sets of blocks or regular expressions, illustrated in Figure 3.

The resource compiler `hunlex` is a standalone program written in OCaml which comes with a command-line as well as a Makefile as toplevel control interface. The internal workings of `hunlex` are as follows.

As the morphological grammar is parsed by the precompiler, rule objects are created. A block is read and parsed into functions which each transform the 'affix-rule' data-structure by enriching its internal representation according to the semantic content of the block. At the end of each unit, the empty rule is passed to the composition of block functions to result in a specific rule. Thanks to OCaml's flexibility of function abstraction and composition, this design makes it easy to implement macros of arbitrary blocks directly as functions. When the grammar is parsed, rules are arranged in a directed (possibly cyclic) graph with edges representing possible rule applications as given by the output specifications.

Precompilation proceeds by performing a recursive closure on this graph starting from lexical nodes. Rules are indexed by 'levels' and contiguous rule-nodes that are on the same level are merged along the edges if constraints on rule application (feature and match conditions, etc.) are satisfied. These precompiled affix-clusters and complex lexical items are to be placed in the `aff` and `dic` file, respectively.

Instead of affix merging, closure between rules *a* and *b* on different levels causes the affix clusters in the closure of *b* to be registered as rules in a hash and their indexes recorded on *a*. After the entire lexicon is read, these index sets registered on rules are considered. The affix cluster rules to be output into the affix file are arranged into maximal subsets such that if two output affix cluster rules *a* and *b* are in the same set, then every item or affix to which *a* can be applied, *b* can also be applied. These sets of affix clusters correspond to partial paradigms which each full paradigm either includes or is disjoint with. The resulting sets of output rules are assigned to a flag and items referencing them will specify the appropriate combination of flags in the output `dic` and `aff` file. Since equivalent affix cluster rules are conflated, the compiled resources are always optimal in the following three ways.

First, the affix file is *redundancy free*: no two affix rules have the same form. With hand-coded affix files this can almost never be guaranteed since one is always inclined to group affix rules by linguistically motivated paradigms thereby possibly duplicating entries. A redundancy-free set of affix rules will enhance performance by minimizing the search space for affixes. Note that conflation of

identical rules by the runtime layer is not possible without reindexing the flags which would be very computationally intensive if done at runtime.

Second, given the redundancy free affix-set, maximizing homogeneous rulesets assigned to a flag *minimizes the number of flags* used. Since the internal representation of flags depends on their number, this has the practical advantage of reducing memory requirements for the runtime layer.

Third, identity of output affix rules is calculated relative to mode and configuration settings, therefore *identical morphs* with different morphological tags *will be conflated* for recognizers (spellchecking) where ambiguity is irrelevant, while for analysis it can be kept apart. This is impossible to achieve without a precompilation stage. Note that finite state transducer-based systems perform essentially the same type of optimizations, eliminating symbol redundancy when two symbols behave the same in every rule, and eliminating state redundancy when two states have the exact same continuations.

Though the bulk of the knowledge used by spellcheckers, by stemmers, and by morphological analysis and generation tools is shared (how affixes combine with stems, what words allow compounding), the ideal resources for these various tasks differ to some extent. Spellcheckers are meant to help one to conform to orthographic norms and therefore should be error sensitive, stemmers and morphological analyzers are expected to be more robust and error tolerant especially towards common violations of standard use. Although this seems at first to justify the individual efforts one has to invest in tailoring one's resources to the task at hand, most of the resource specifics are systematic, and therefore allow for automatic fine-tuning from a central knowledge base. Configuration within `hunlex` allows the specification of various features, among others:

- selection of registers and degree of normativity based on usage qualifiers in the database (allows for boosting robustness for analysis or stick to normativity for synthesis and spellchecking)
- flexible selection of output information: choice of tagset for different encodings, support for sense indexes
- arbitrary selection of morphemes
- setting levels of morphemes (grouping of morphs that are precompiled as a cluster to be stripped with one rule application by the runtime layer)
- fine-tuning which morphemes are stripped during stemming
- arbitrary selection of morphophonological features that are to be observed or ignored (allows for enhancing robustness by e.g., tolerating non-standard regularizations)

The input description language allows for arbitrary attributes (ones encoding part of speech, origin, register, etc.) to be specified in the description. Since any set of attributes can be selected to be compiled into the runtime resources, it takes no more than precompiling the central database with the appropriate configuration for the runtime analyzer to be used as an arbitrary word annotation tool, e.g., style annotator or part of speech tagger. We also provide an implementation of a feature-tree based tag language which we successfully used for the description of Hungarian morphology.

If the resources are created for some filtering task, say, extracting (possibly inflected) proper nouns in a text, resource optimization described above can save considerable amounts of time compared to full analysis followed by post-processing. While the relevant portion of the dictionary might be easily filtered therefore speeding up lookup, tailoring a corresponding redundancy-free affix file would be a hopeless enterprise without the precompiler.

As we mentioned, our offline layer can be configured to cluster any or no sets of affixes together on various levels, and therefore resources can be optimized for either memory use (affix by affix stripping) or speed (generally toward one level stripping). This is a major advantage given potential applications as diverse as spellchecking on the word processor of an old 386 at one end, and

industrial scale stemming on terabytes of web content for IR at the other.

In sum, our offline layer allows for the principled maintenance of a central resource, saving the redundant effort that would otherwise have to be invested in encoding very similar knowledge in a task-specific manner for each word level analysis task.

3 Conclusion

The importance of word level analysis can hardly be questioned: spellcheckers reach the extremely wide audience of all word processor users, stemmers are used in a variety of areas ranging from information retrieval to statistical machine translation, and for non-isolating languages morphological analysis is the initial phase of every natural language processing pipeline.

Over the past decades, two closely intertwined methods emerged to handle word analysis tasks, affix stripping and finite state transducers (FSTs). Since both technologies can provide industrial strength solutions for most tasks, when it comes to choice of actual software and its practical use, the differences that have the greatest impact are not lodged in the algorithmic core. Rather, two other factors play a role: the ease with which one can integrate the software into applications and the infrastructure offered to translate the knowledge of the grammarian to efficient and maintainable computational blocks.

To be sure, in an end-to-end machine learning paradigm, the mundane differences between how the systems interact with the human grammarians would not matter. But as long as the grammars are written and maintained by humans, an offline framework providing a high-level language to specify morphologies and supporting configurable precompilation that allows for resource sharing across word-analysis tasks addresses a major bottleneck in resource creation and management.

The Xerox Finite State Toolkit provides comprehensive high-level support for morphology and lexicon development (Beesley and Karttunen, 2003). These descriptions are compiled into minimal deterministic FST-s, which give excellent runtime performance and can also be extended to error-tolerant analysis for spellchecking Oflazer (1996). Nonetheless, XFST is not free software, and as long as the work is not driven by academic curiosity alone, the LGPL-style license of our toolkit, explicitly permitting reuse for commercial purposes as well, can already decide the choice.

There are other free open source analyzer technologies, either stand-alone analyzers such as the Stuttgart Finite State Toolkit (SFST, available only under the GPL, see `www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html`, Smid et al. (2004)) or as part of a powerful integrated NLP platform such as Intex/NooJ (freely available for academic research to individuals affiliated with a university only, see `intex.univ-fcomte.fr`; a clone called Unitex is available under LGPL, see `www-igm.univ-mlv.fr/~unitex`.) Unfortunately, NooJ has its limitations when it comes to implementing complex morphologies (Vajda et al., 2004) and SFST provides no high-level offline component for grammar description and configurable resource creation.

We believe that the liberal license policy and the powerful offline layer contributed equally to the huge interest that our project generated, in spite of its relative novelty. `MySpell` was not just our choice: it is also the spell-checking library incorporated into OpenOffice.org, a free open-source office suite with an ever wider circle of users. The Hungarian build of OpenOffice is already running our C++ runtime library, but OpenOffice is now considering to completely replace `MySpell` with our code. This would open up the possibility of introducing morphological analysis capabilities in the program, which in turn could serve as the first step towards enhanced grammar checking and hyphenation.

Though in-depth grammars and lexica are available for nearly as many languages in FST-based frameworks (InXight Corporation's LinguistX platform supports 31 languages), very little of this material is available for grammar hacking or open source dictionary development. In addition to permissive license and easy to integrate infrastructure, the fact that the `hunmorph` routines

are backward compatible with already existing and freely available spellchecking resources for some 50 languages goes a long way toward explaining its rapid spread.

For Hungarian, `hunlex` already serves as the development framework for the MORPHDB project which merges three independently developed lexical databases by critically unifying their contents and supplying it with a comprehensive morphological grammar. It also provided a framework for our English morphology project that used the XTAG morphological database for English (see `ftp.cis.upenn.edu/pub/xtag/morph-1.5`, Karp et al. (1992)). A project describing the morphology of the Beás dialect of Romani with `hunlex` is also under way.

The `hunlex` resource precompiler is not architecturally bound to the `aff/dic` format used by our toolkit, and we are investigating the possibility of generating FST resources with it. This would decouple the offline layer of our toolkit from the details of the runtime technology, and would be an important step towards a unified open source solution for method-independent resource development for word analysis software.

Acknowledgements

The development of `hunmorph` and `hunlex` is financially supported by the Hungarian Ministry of Informatics and Telecommunications and by MATÁV Telecom Co., and is led by the Centre for Media Research and Education at the Budapest University of Technology. We would like to thank the anonymous reviewers of the ACL Software Workshop for their valuable comments on this paper.

Availability

Due to the confligting needs of Unix, Windows, and MacOs users, the packaging/build environment for our software has not yet been finalized. However, a version of our tools, and some of the major language resources that have been created using these tools, are available at `mokk.bme.hu/resouces`.

References

Kenneth R. Beesley and Lauri Karttunen. 2003. *Finite State Morphology*. CSLI Publications.

Daniel Karp, Yves Schabes, Martin Zaidel, and Dania Egedi. 1992. A freely available wide coverage morphological analyzer for english. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING-92) Nantes, France.*

László Németh, Viktor Trón, Péter Halácsy, András Kornai, András Rung, and István Szakadát. 2004. Leveraging the open-source ispell codebase for minority language analysis. In *Proceedings of SALTMIL 2004*. European Language Resources Association. URL `http://www.lrec-conf.org/lrec2004`.

Kemal Oflazer. 1996. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1):73–89.

James Lyle Peterson. 1980. *Computer programs for spelling correction: an experiment in program design*, volume 96 of *Lecture Notes in Computer Science*. Springer.

Helmut Smid, Arne Fitschen, and Ulrich Heid. 2004. SMOR: A German computational morphology covering derivation, composition, and inflection. In *Proceedings of the IVth International Conference on Language Resources and Evaluation (LREC 2004)*, pages 1263–1266.

Péter Vajda, Viktor Nagy, and Emília Dancsecs. 2004. A Ragozási szótártól a NooJ morfológiai moduljáig [from a morphological dictionary to a morphological module for NooJ]. In *2nd Hungarian Computational Linguistics Conference*, pages 183–190.