# Compiling Apertium morphological dictionaries with HFST and using them in HFST applications

**Tommi A Pirinen, Francis M. Tyers**

University of Helsinki, Universitat d'Alacant
FI-00014 University of Helsinki Finland, E-03071 Alacant Spain
tommi.pirinen@helsinki.fi, ftyers@dlsi.ua.es

**Abstract**

In this paper we aim to improve interoperability and re-usability of the morphological dictionaries of Apertium machine translation system by formulating a generic finite-state compilation formula that is implemented in HFST finite-state system to compile Apertium dictionaries into general purpose finite-state automata. We demonstrate the use of the resulting automaton in FST-based spell-checking system.

**Keywords:** finite-state, dictionary, spell-checking

## 1. Introduction

Finite-state automata are one of the most effective format for representing natural language morphologies in computational format. The finite-state automata, once compiled and optimised via process of minimisation are very effective for parsing running text. This format is also used when running morphological dictionaries in machine-translation system Apertium (Forcada et al., 2011)[1]. In this paper we propose a generic compilation formula to compile the dictionaries into weighted finite state automata for use with any FST tool or application. We implement this system using a free/libre open-source finite-state API HFST (Lindén et al., 2011)[2]. HFST is a general purpose programming interface using a selection of freely-available finite-state libraries for the handling of finite-state automata. While Apertium uses the dictionaries and the finite-state automata for machine translation, HFST is used in multitude of other applications ranging from basic morphological analysis (Lindén et al., 2011) to end-user applications such as spell-checking (Pirinen and Lindén, 2010) and predictive text-entry for mobile phones (Silfverberg et al., 2011). In this article we show how to generate automatically a spell-checker from an Apertium dictionary and evaluate roughly the usability of the automatically generated spell-checker. The rest of the article is laid out as follows: In section 2. we describe the generic compilation formula for the HFST-based compilation of Apertium dictionaries and the formula for induction of spell-checkers error model from Apertium's dictionary. In section 3. we introduce the Apertium dictionary repository and the specific dictionaries we use to evaluate our systems. In section 4. we evaluate speed and memory usage of compilation and application of our formula against Apertium's own system and show that our system has roughly same coverage and explain the differences arise from.

## 2. Methods

The compilation of Apertium dictionaries is relatively straight-forward. We assume here standard notations for finite-state algebra. The morphological combinatorics of Apertium dictionaries are defined in following terms: There is one set of root morphs (finite strings) and arbitrary number of named sets of affix morphs called `pardefs`. Each set of affix morphs is associated with a name. Each morph can also be associated with a paradigm reference pointing to a named subset of affixes. As an example, a language of singular and plural of *cat* and *dog* in English would be described by root dictionary consisting of morphs `cat` and `dog`, both of which point on the right-hand side to pardef named `number`. The number affix morphs are defined then as set of two morphs, namely `s` for plural marker and empty string for singular marker.

Each morph can be compiled into single-path finite-state automaton[3] containing the actual morph as string of UTF-8 arcs $m$. The morphs in the root dictionary are extended from left or right sides by joiner markers iff they have a pardef definition there and each affix dictionary is extended on the left (for suffixes) or right (for prefixes) by the pardef name marker. In the example of *cats, dogs* language this would mean finite state paths `c a t NUMBER`, `d o g NUMBER`, `NUMBER s` and `NUMBER` $\epsilon$, where $\epsilon$ as usual marks zero-length string[4]. These sets of roots and affixes can be compiled into disjunction of such joiner delimited morphs. Now, the morphotactics can be defined as related to joiners by any such path that contains joiners only as pairs of adjacent identical paradigm references, such as `c a t NUMBER NUMBER s` or `d o g NUMBER NUMBER` $\epsilon$, but not `c a t NUMBER d o g NUMBER` or `NUMBER s NUMBER`

[1] http://www.apertium.org

[2] http://hfst.sf.net

[3] the full formula allows any finite-state language as morph, compiled from regular expressions, the extension to this is trivial but for readability we present the formula for string morphs

[4] In the current implementation we have used temporarily a special non-epsilon marker as this decreases the local indeterminism and thus compilation time

s. The finite-state formula for this morphotactics is defined by

$$M_x = (\Sigma \cup \bigcup_{x \in p} xx)^\star, \tag{1}$$

where $p$ is set of pardef names and $\Sigma$ the set of symbols in morphs not including the set of pardef names. Now the final dictionary is simply composition of these morphotactic rules over the repetion of affixes and roots:

$$(M_a \cup M_r)^\star \circ M_x, \tag{2}$$

where $M_a$ is the disjunction of affixes with joiners, $M_r$ the disjunction of roots with joiners, and $M_x$ the morphotactics defined in formula 1. This is a variation of morphology compilation formula presented in various HFST documentation, such as (Lindén et al., 2011).

### 2.1. Implementation Details

There are lot of finer details we will not thoroughly cover in this article, as they are mainly engineering details. In this section we shortly summarise specific features of HFST-based FST compilation that result in meaningful differences in automaton structure or working. One of the main source of differences is that HFST automata are two-sided and compiled only ones from the source code whereas Apertium generates two different automata for analysis and generation. In these automata the structure may be different, since Apertium dictionaries have ways of marking morphs limited to generation or analysis only, so they will only be included in one of the automatons. Our approach to this is to use special symbols called flag-diacritics (Beesley and Karttunen, 2003) to limit the paths as analysis only or generation only on runtime, but still including all paths in the one transducer that gets compiled.

Another main difference in processing comes from the special word-initial, word-final and separate morphs that in Apertium are contained in separate automata altogether, but HFST tools do not support use of multiple automata for analysis, so these special morphs will be concatenated optionally to beginning or end of the word, or disjuncted to the final automata respectively. These special morphs include things like article *l'* in French as bound form.

### 2.2. Creating a Spell-Checker Automatically

To create a finite-state spell-checker we need two automata, one for the language model, for which the dictionary compiled as described earlier will do, and one for the error model (Pirinen and Lindén, 2010). A classic baseline error model is based on the edit distance algorithm (Levenshtein, 1966; Damerau, 1964), that defines typing errors of four types: pressing extra key (insertion), not pressing a key (deletion), pressing wrong key (change) and pressing two keys in wrong order (swap). There have been many finite-state formulations of this, we use the one defined in (Schulz and Mihov, 2002; Pirinen and Lindén, 2010). The basic version of this where the typing errors of each sort have equal likelihood for each letters can be induced from the compiled language model, and this is what we use in this paper. The induction of this model is relatively straightforward; when compiling the automaton, save each unique UTF-8 codepoint found in the morphs[5]. For each character generate the identities in start and end state to model correctly typed runs. For each of the error types the generate one arc from initial state to the end state modelling that error, except for swap which it requires one auxiliary state for each character pair.

## 3. Materials

The Apertium project hosts a large number of morphological dictionaries for each of the languages translated. From these we have selected three dictionaries to be tested: Basque from Basque-Spanish pair as it is released dictionary with the biggest on-disk size, Norwegian Nynorsk from the Norwegian pair as a language that has some additional morphological complexity, such as compounding, and Manx from as a language that currently lacks spell-checking tools to demonstrate the plausibility of automatic conversion of Apertium dictionary into a spell-checker[6].

To evaluate the use of resulting morphological dictionaries and spell-checkers we use following Wikipedia database dumps[7]: `euwiki-20120219-pages-articles.xml.bz2`, `nnwiki-20120215-pages-articles.xml.bz2`, and `gvwiki-20120215-pages-articles.xml.bz2`. For the purpose of this article we performed very crude cleanup and preprocessing to Wikipedia data picking up the text elements of the article and discarding most of Wikipedia markup naïvely[8].

## 4. Test Setting and Evaluation

To get one view on differences made by generic compilation formula instead of direct automata building used by Apertium we look at the created automata, this will also give us a rough idea of what its efficiency might be. In table 1 we give the counts of nodes and edges, in that order, in the graphs compiled from the dictionaries. Note, that in case of Apertium it is the sum of all the separate automata states and edges that is counted. The small differences in sizes of graphs are mostly caused by the different handling of generation vs. analysis mode. The difference in sizes of automata on disk in is shown in table 2. The size of HFST au-

[5] The description format of Apertium requires declaration of exemplar character set as well, but as this is only used in the tokenisation algorithm (Garrido-Alenda et al., 2002) , which is not going to be used, we induce the set from the morphs

[6] We also provide a Makefile script to recreate results of this article for any language in Apertium's repository

[7] `http://download.wikipedia.org/`

[8] For details see the script in `http://hfst.svn.sourceforge.net/viewvc/hfst/trunk/lrec-2011-apertium/`.

tomata can be attributed to the clever compression algorithm used by HFST (Silfverberg and Lindén, 2009).

| **Lang.** | **Apertium** `LR` | **Apertium** `RL` | **HFST** |
|---|---:|---:|---:|
| Basq. | 30,114 | 34,005 | 34,824 |
| | 59,321 | 68,030 | 68,347 |
| Norg. | 56,226 | 55,722 | 56,871 |
| | 138,217 | 132,475 | 139,259 |
| Manx | 13,055 | 12,955 | 12,920 |
| | 28,220 | 27,062 | 27,031 |

**Table 1:** Size of HFST-based system against original (count of nodes first, then edges)

| **Lang.** | **Apertium** `LR` | **Apertium** `RL` | **HFST** |
|---|---:|---:|---:|
| Basq. | 252 KiB | 289 KiB | 1,7 MiB |
| Norg. | 558 KiB | 535 KiB | 3,7 MiB |
| Manx | 108 KiB | 110 KiB | 709 KiB |

**Table 2:** Size of HFST-based system against original (as B on disk)

To test efficiency we measure times of running various tasks. The times and memory usage have been measured using GNU `time` utility and `getrusage` system call's `ru_utime` field, averaged over three test runs. The tests were performed on quad-core Intel Xeon E5450 @ 3.00 GHz with 64 GiB of RAM.

First we measure speed of analysing a full corpus with the result automaton. The speed is measured in the table 3, in seconds to precision that was available in our system. Curiously the results do not give direct advantage to either of the system but it seems to depend on the language which system is a better choice for corpus analysis.

| **Language** | **Apertium** | **HFST** |
|---|---:|---:|
| Basque | 32.0 s | 18.4 s |
| Norwegian | 2.4 s | 5.5 s |
| Manx | 1.6 s | 2.2 s |

**Table 3:** Speed of HFST-based system against original in corpus analysis (as s in user time)

Similarly we measure the speed of current compilation process in table 4. In here there's an obvious advantage to manual building of the automaton (see (Rojas et al., 2005) for the precise algorithm used) over the finite-state algebra method, as is in line with earlier results for lexc building in (Lindén et al., 2009).

Finally we evaluate the usability of dictionaries meant for machine translation as spell-checkers by running the finite-state spell checkers we produced automatically through a large corpus and show the measure both speed and quality of the results. The errors were automatically generated to Wikipedia text's correct words using simple algorithm that may generate one Levenshtein error per each character position at probability of $\frac{1}{33}$. This test shows only rudimentary results on the plausibility of using machine translation dictionary for spell-checking; for more thorough evaluation of efficiency of finite-state spell-checking see (Hassan et al., 2008).

| **Language** | **Apertium time** | **HFST time** |
|---|---:|---:|
| Basque | 35.7 s | 160.0 s |
| Norwegian | 6.6 s | 200.2 s |
| Manx | 0.8 s | 11.2 s |

**Table 4:** Speed of HFST-based system against original in compilation (as seconds of user time)

| **Language** | **Speed (words/sec)** |
|---|---:|
| Basque | 7,900 |
| Norwegian | 9,200 |
| Manx | 4,700 |

**Table 5:** Efficiency of spelling correction in artificial test setup, average over three runs.

## 5. Conclusions

In this article we have shown a general formula to compile morphological dictionaries from machine-translation system Apertium in generic FST system of HFST and using the result in HFST-based application of spell-checking.

## 6. Future Work

In this article we showed a basic method to gain more inter-operability between generic FST system of HFST and a specialised morphological dictionary writing formalism of machine-translation system Apertium by implementing a generic compilation formula to compile the language descriptions. In future research we are leveraging this and other related formulas into automatic optimisation of the final automata using the information present in the language description to optimise instead of relying generic graph algorithms for the final minimised result automata.

We demonstrated importing the compiled dictionary as a language model and inducing error model for real-world spell-checking applications. Further development in this direction should aim for interoperable formalisms, formats and mechanisms for language models and end applications of all relevant language technology tools.

## Acknowledgements

We thank the HFST and Apertium contributors for fruitful internet relayed chats, and the two anonymous reviewers for their helpful suggestions.

## 7. References

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

F J Damerau. 1964. A technique for computer detection and correction of spelling errors. *Commun. ACM*, (7).

Mikel L. Forcada, Mireia Ginestí-Rosell, Jacob Nordfalk, Jim O'Regan, Sergio Ortiz-Rojas, Juan Antonio Pérez-Ortiz, Felipe Sánchez-Martínez, Gema Ramírez-Sánchez, and Francis M. Tyers. 2011. Apertium: a free/open-source platform for rule-based machine translation. *Machine Translation*, jul.

Alicia Garrido-Alenda, Mikel L. Forcada, and Rafael C. Carrasco. 2002. Incremental construction and maintenance of morphological analysers based on augmented letter transducers. In *Proceedings of TMI 2002 (Theoretical and Methodological Issues in Machine Translation, Keihanna/Kyoto, Japan)*, pages 53–62.

Ahmed Hassan, Sara Noeman, and Hany Hassan. 2008. Language independent text correction using finite state automata. In *Proceedings of the Third International Joint Conference on Natural Language Processing*, volume 2, pages 913–918.

V. I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics—Doklady 10, 707–710. Translated from Doklady Akademii Nauk SSSR*, pages 845–848.

Krister Lindén, Miikka Silfverberg, and Tommi Pirinen. 2009. Hfst tools for morphology—an efficient open-source package for construction of morphological analyzers. In Cerstin Mahlow and Michael Piotrowski, editors, *sfcm 2009*, volume 41 of *Lecture Notes in Computer Science*, pages 28—47. Springer.

Krister Lindén, Miikka Silfverberg, Erik Axelson, Sam Hardwick, and Tommi Pirinen, 2011. *HFST—Framework for Compiling and Applying Morphologies*, volume Vol. 100 of *Communications in Computer and Information Science*, pages 67–85. Springer.

Tommi A Pirinen and Krister Lindén. 2010. Finite-state spell-checking with weighted language and error models. In *Proceedings of the Seventh SaLTMiL workshop on creation and use of basic lexical resources for less-resourced languagages*, pages 13–18, Valletta, Malta.

Sergio Ortiz Rojas, Mikel L. Forcada, and Gema Ramírez Sánchez. 2005. Construcción y minimización eficiente de transductores de letras a partir de diccionarios con paradigmas. *Procesamiento del Lenguaje Natural*, (35):51–57.

Klaus Schulz and Stoyan Mihov. 2002. Fast string correction with levenshtein-automata. *International Journal of Document Analysis and Recognition*, 5:67–85.

Miikka Silfverberg and Krister Lindén. 2009. Hfst runtime format—a compacted transducer format allowing for fast lookup. In Bruce Watson, Derrick Courie, Loek Cleophas, and Pierre Rautenbach, editors, *FSMNLP 2009*, 13 July.

Miikka Silfverberg, Mirka Hyvärinen, and Tommi Pirinen. 2011. Improving predictive entry of finnish text messages using irc logs. pages 69–76.