Using HFST for Creating Computational Linguistic Applications*

Krister Lindén, Erik Axelson, Senka Drobac,
Sam Hardwick, Miikka Silfverberg, and Tommi A Pirinen

University of Helsinki
Department of Modern Languages
Unioninkatu 40 A
FI-00014 Helsingin yliopisto, Finland
{krister.linden, erik.axelson, senka.drobac, sam.hardwick,
miikka.silfverberg,tommi.pirinen}@helsinki.fi

Abstract. HFST – Helsinki Finite-State Technology (`hfst.sf.net`) is a framework for compiling and applying linguistic descriptions with finite-state methods. HFST currently collects some of the most important finite-state tools for creating morphologies and spellcheckers into one open-source platform and supports extending and improving the descriptions with weights to accommodate the modeling of statistical information. HFST offers a path from language descriptions to efficient language applications. In this article, we focus on aspects of HFST that are new to the end user, i.e. new tools, new features in existing tools, or new language applications, in addition to some revised algorithms that increase performance.

Keywords: finite-state applications, morphology, tagging, HFST

1 Introduction

[1] HFST – Helsinki Finite-State Technology (`hfst.sf.net`) is designed for creating and compiling morphologies, which has been documented in, e.g., [14,11]. In this article we focus on the applications created with HFST and some of their theoretical motivations. HFST contains open-source replicas of xfst, lexc and twolc which are well-known and well-researched tools for morphology building, see [3]. The tools support both parallel and cascaded application of transducers. This is outlined in Section 2 along with some new and previously undocumented extensions to the Xerox tools in HFST.

There are a number of tools for describing morphologies. Many of them start with the item-and-arrangement approach in which an arrangement of sublexicons contain lists of items that may continue in other sublexicons. A formula for compiling such lexical descriptions was documented in [14]. In Section 3, we demonstrate a simplified procedure for how such morphotactic descriptions can

[1] Author's preprint version, may differ from the print version

be compiled into finite-state lexicons using finite-state operations. To realize the morphological processes, rules may be applied to the finite-state lexicon. In addition, HFST now also offers the capability to train and apply part-of-speech taggers on top of the morphologies using parallel weighted finite-state transducers on text corpora, which is outlined and evaluated in Section 4.

Using compiled morphologies, a number of applications have been created, e.g. spellcheckers for close to 100 languages and hyphenators for approximately 40 different languages. The spellcheckers were derived from open-source dictionaries and integrated with OpenOffice and LibreOffice, e.g. a full-fledged Greenlandic spellchecker, which is a polyagglutinative language, is currently available for OpenOffice via HFST. By adding the tagger capability, we have also created an improved spelling suggestion mechanism for words in context. The spellchecker applications and some of their theoretical underpinnings are described in Section 5.

Finally in Section 6, we discuss some additional applications such as synonym and translation dictionaries as well as a framework for recognizing multi-word expressions for information extraction and how this can be done using finite-state technologies.

2 Building Morphologies

One of the earliest and most important goals of the HFST project has been to provide open-source utilities for compiling full-fledged morphological analyzers, which may be used for constructing spellcheckers, taggers and other language technological utilities of good quality. The Xerox toolkit [3] is among the most widely used frameworks for constructing morphological utilities. The toolkit is used to compile linguistic description into morphological analyzers. More specifically, Xerox tools include the finite-state lexicon compiler lexc, the two-level rule compiler `twolc` and the cascading replace rule compiler xfst. HFST includes the tools **hfst-lexc**, **hfst-twolc** and **hfst-xfst**, which provide full backward compatibility with Xerox tools and augment their functionality.

Lexicon files for the lexicon compiler **hfst-lexc** describe the morphotactics of a language using sub-lexicons containing lists of morphs. A rule component can be used for realizing phonological changes occurring within the morphs or at their boundaries. In this article, we do not introduce new features of **hfst-lexc**. Adding weights to lexicons using **hfst-lexc** is described in [13].

The rule compilers **hfst-xfst** and **hfst-twolc** provide almost the same functionality. Both are used to realize morphophonological variations on a lexicon compiled with **hfst-lexc**. The difference between the tools is that **hfst-xfst** rules are applied in succession gradually altering the lexicon whereas **hfst-twolc** rules are applied as parallel constraints limiting the realizations of morphophonemes used in the **hfst-lexc** lexicon.

2.1 Parallel Rules with Negative Contexts

The rule compiler **hfst-twolc** provides backward compatibility with the Xerox tool twolc, but it augments twolc functionality by providing a new type of rules. In **hfst-twolc**, rules can have negative contexts.

```
! Change x to y after the string "a b" unless "d" or
! "c d" follows.
"Rule with negative context"
x:y <=> a b _     ;
        except
            _ ( c ) d ;

"Rule without negative context"
x:y <=> a b _ [ ? - [ c | d ] | c [ ? - d ] | .#. ] ;
```

Fig. 1. Negative context rule and corresponding traditional two-level rule

In traditional two-level rules, it can be very difficult to express that a certain alternation should not occur in a given context. Such restrictions are required because of rule conflicts, i.e. clashes of two or more contradicting rules. When a rule conflict occurs, the contexts of some of the contradicting rules need to be restricted in order to resolve the conflict.[2]

Sometimes an automated mechanism called conflict resolution of two-level rules [24] can be used to resolve rule conflicts, but conflict resolution works only if the conflicting rules can be ordered into chains of subcases. Often this can be difficult to accomplish, especially for grammar writers who are not especially well acquainted with writing regular expressions.

In twolc syntax, prohibition rules such as `x:y /<= C1 _ C2 ;` can be used to forbid a pair `x:y` in a context where the left context matches the regular expression `C1` and the right context matches `C2`. Unfortunately prohibition rules do not solve the problem of conflicting rules, because they do not participate in conflict resolution and simply adding new rules to a two-level grammar does not remove rule conflicts.[3]

Using an extension to the Xerox two-level rule syntax in `hfst-twolc`, it is possible to formulate rules with negative contexts, i.e. contexts that prohibit the triggering of the rule. Figure 1 shows a schematic example of a negative context rule and a traditional Xerox style rule, which has the same effect as the rule with the negative context.

[2] `http://www.cis.upenn.edu/~cis639/docs/twolc.html`

[3] Note that, if prohibition rules would participate in conflict resolution, it would still be challenging to write them in such a manner that conflict resolution could apply.

Rules with negative contexts are used in the Kyrgyz morphology developed in the Apertium project[4]. They significantly shorten the two-level grammar[5].

2.2 Cascaded Rules: Explanations and Examples

HFST replace rules provide backward compatibility with XFST replace rules, described in [10,3]. Although they mostly share the same notation, the HFST replace rules differ in that they were compiled with the `.r-glc.` operator [7] and preference relations described in [29]. This approach makes it possible to more freely define contexts in parallel rules and to easily add new functionalities in future.

We present a general account of using replace rules with the command-line tool `hfst-regexp2fst`. Since these rules mostly follow the behaviour of the XFST rules, for a detailed description of each rule, see The Finite State Morphology book [3].

Simple rules. A simple right arrow replace rule,

$$A \rightarrow B \tag{1}$$

where A and B are regular expressions, expresses that A in the upper language maps to B in the lower language.

Replace rules can be compiled into a transducer using `hfst-regexp2fst`. This tool takes a regular expresion as input and gives a corresponding transducer written in a binary file as output. To convert transducers from binary format to text (ATT) format, there is an HFST tool `hfst-fst2txt`. Therefore, the upper rule could be compiled as in Figure 2.

```
$ echo "A -> B ;" | hfst-regexp2fst | hfst-fst2txt
```

Fig. 2. Compiling simple replace rule

The regular expression can be read from a file, or saved to a file (Figure 3).

```
$ echo "A -> B ;" > regex
$ hfst-regexp2fst -i regex -o transducer
$ hfst-fst2txt transducer
```

Fig. 3. Reading from, and writing to a file

Rules can be applied to any language by using composition. The result of composing a word `ABCD` with the aforementioned rule `A -> B`, the result is the

[4] `http://www.apertium.org/`
[5] Personal communication with Francis Tyers.

transducer in Figure 4. It has 5 states (0 – 4), and only state 4 is final. The transitions go from a state in the first column to a state in the second column. The input label of a transition is displayed in the third column and the output label in the fourth. The default output of `hfst-regexp2fst` is weighted. Since weights are not used in replace rules, the weights get the value 0.0 when the rule is compiled. The weights are displayed in the final column.

```
$ echo "A B C D .o. A -> B ;" | hfst-regexp2fst | hfst-fst2txt
0       1       A       B       0.000000
1       2       B       B       0.000000
2       3       C       C       0.000000
3       4       D       D       0.000000
4       0.000000
```

Fig. 4. Composing a rule with a word

Context. Every rule can have a context in which the replacement is made. Here, A will be mapped to B if and only if it is between regular expressions L and R.

$$A \rightarrow B \mid\mid L\_ \; R \tag{2}$$

Also, multiple context pairs are supported, when separated by comma.

$$A \rightarrow B \mid\mid L_1\_ \; R_1, \; \ldots \; , \; L_i\_ \; R_i \tag{3}$$

```
$ echo " m a n a a b .o. a -> b || m _ n , _ b;" |
hfst-regexp2fst | hfst-fst2txt
0       1       m       m       0.000000
1       2       a       b       0.000000
2       3       n       n       0.000000
3       4       a       a       0.000000
4       5       a       b       0.000000
5       6       b       b       0.000000
6       0.000000
```

Fig. 5. Replace rule between two contexts

In Figure 5, the rule expression says that `a` in the upper language is mapped to `b` in lower language when between `m` and `n`, or in front of `b`. In the word `manaab`, the first and last `a` are mapped to `b`, because they occur between corresponding contexts, but the second `a` is kept unchanged.

In a replace rule where A is the upper and B the lower language, there are four contextual directions that can be used with replace rules. For example,

Table 1. Different context directions in Replace Rules

Operator	Operator description
`||`	both contexts are taken from the upper language
`//`	the left context is taken from the lower language, the right from the upper
`\\`	the left context is taken from the upper language, the right from the lower
`\/`	both contexts are taken from the lower language.

when the `//` sign is used as context orientation operator, the left context will be taken from the lower language. It is thus possible for the replace function to write its own context. This is shown in Figure 6.

```
$ echo " b a a .o. a -> b // b _ ;" | hfst-regexp2fst | hfst-fst2txt
0       1       b       b       0.000000
1       2       a       b       0.000000
2       3       a       b       0.000000
3       0.000000
```

Fig. 6. Replace rule writes its own context

Other replace functions. We have hitherto only used the right arrow replace operator, but there are many other operators that can be used. All the operators listed in Table 2 have their left arrow version, which is the inversion of the right operator. Furthermore, all the rules can be used with epenthesis `[. .]` and markup `[...]` operators. The epenthesis operator should be used with empty strings to avoid replacing infinitely many epsilons, while the markup operator is used to insert markers around a word.

Table 2. List of right replace operators that can be used in HFST

Right replace operators	Replace function
`->`	Replace
`(->)`	Replace optional
`@->`	Longest match from left to right
`->@`	Longest match from right to left
`@>`	Shortest match from left to right
`>@`	Shortest match from right to left

Parallel rules. Parallel rules are used when there are multiple replacements at the same time. The general parallel replace rule expression consists of individual replace rules separated by two commas.

$$A_1 \rightarrow\ B_1 \,||\, L_1 \_\ R_1\ ,,\ \ldots\ ,,\ A_i \rightarrow\ B_i \,||\, L_i \_\ R_i \tag{4}$$

```
$ echo " a a a .o. [.a*.] @-> b ;" | hfst-regexp2fst > a.fst
$ echo " b a a b .o. a @-> %[ ... %] ;" | hfst-regexp2fst > b.fst
```

Fig. 7. Epenthesis and markup operators

In XFST, all rules in a parallel rule expression have to have the same format, i.e. they need to have the same arrow and the same context layout. In HFST, the constraint that all the rules should have the same arrow is kept, but the context layout can differ freely. Therefore, the rule expression in Figure 8 would not be allowed in XFST, but is valid in HFST.

```
$  echo " a -> b || m _ n ,, c -> d ;" | hfst-regexp2fst | hfst-fst2txt
```

Fig. 8. Parallel rules with different context layouts

3 Morphological Descriptions

Popular formalisms for describing the morphotactics of a language tend to be some variation of the item and arrangement scheme. We build on this to generalize from `lexc` into other item-and-arrangement notations such as Hunmorph and Apertium. This gives us the option of describing morphology in a notation that is compiled into finite-state transducers which we can continue to process with other HFST tools. We demonstrate how compilation could proceed when reducing a `lexc` lexicon into a sequence of finite-state operations. A morphological programming language like `SFST-PL` forgoes a standard lexicon interface like `lexc` and only offers the end user an option to construct the lexicon using finite-state operations.

3.1 Morphotax and Morphological Formulæ

Morphotax is the component dealing with morphological combinations in language description. This concerns word-formation processes, such as affixation and compounding. There are numerous formalisms for describing morphotactics in natural language processing applications, such as hunspell[6] (and its older *spell relatives), the Apertium lttoolbox[7] or the Xerox lexc [3]. The HFST toolset contains parsers and compilers for reading descriptions of morphologies written in these formalisms, which can be used for compiling a finite-state automaton for the other HFST tools to process [20,14].

[6] `http://hunspell.sf.net`
[7] `http://apertium.sf.net`

Consider a trivial lexicon consisting of the English nouns *cat* and *ax* with the empty string as the singular marker and *s* and *es* as plural markers, respectively, forming a lexicon as outlined in Figure 9 using lexc notation.

```
LEXICON Root
Nouns ;

LEXICON Nouns
cat NumberS ;
ax NumberES ;

LEXICON NumberS
# ;
s # ;

LEXICON NumberES
# ;
es # ;
```

Fig. 9. Lexicon in lexc notation

Many contemporary notations for describing morphotactics tend to use the same item-and-arrangement paradigm for combining sets of morphs or morphemes, i.e. they use sublexicons for lists of morphs that may have continuations in other sublexicons. In its most general form, such a paradigm can be defined using a combination of two components expressing local restrictions: (1) a disjunction of morphs in local lexical context repeated infinitely and (2) a morphotactic filter describing the permitted sequence of sublexicons. Both the disjunction of morphs and the morphotactic filter can be described in finite-state form.

In Figure 10, we demonstrate how the lexicon in Figure 9 is compiled into a loop of disjunctions of any of the morphs in the lexicon. Note that we bracket special symbols with at-signs '@'. In this case, we use special joiner symbols to calculate the morphotactics, and the at-signs give the user a hint that they are not supposed to end up in the final result. The sublexicon symbols like `@NumberES@` are used for lining up the sublexicons, and the symbol `@LEX@` is used as a boundary symbol, which simplifies the filter algorithm[8].

In Figure 11, we show the command-line simulation for creating the morphotactic filter and how this filter is composed with the disjunction of morphs to create the final lexicon[9]. In effect, we create a filter that ensures that each

[8] In [14], the algorithm without a separate boundary symbol requires a term complement of the disjunctive closure of the sublexicon symbols.

[9] Whole process is also available from our SVN in `https://hfst.svn.sf.net/svnroot/hfst/trunk/cla-2012-article/morphtest.bash`

```
echo "@Root@ @Nouns@ @LEX@" > strings
echo "@Nouns@ c a t @NumberS@ @LEX@" >> strings
echo "@Nouns@ a x @NumberES@ @LEX@" >> strings
echo "@NumberS@ @END@ @LEX@" >> strings
echo "@NumberS@ s @END@ @LEX@" >> strings
echo "@NumberES@ @END@ @LEX@" >> strings
echo "@NumberES@ e s @END@ @LEX@" >> strings
hfst-strings2fst --has-spaces --disjunct-strings < strings  |
    hfst-repeat -f 1 > bag_of_morphs
```

Fig. 10. Compiling a disjunction of all the morphs in the lexicon

morph, e.g. *es*, only follows the morph that was asking for it on the right, i.e. we require that the same sublexicon symbol, e.g. `@NumberES@`, occurs on both sides of the boundary symbol `@LEX@`. For a full algorithm, see e.g. [14].

```
echo "%@Root%@ [? - %@LEX%@]*"  |  hfst-regexp2fst > start
echo "%@END%@ %@LEX%@"  |  hfst-regexp2fst > end

echo "%@NumberS%@ %@LEX%@ %@NumberS%@ [? - %@LEX%@]*"  |
    hfst-regexp2fst > cont1
echo "%@NumberES%@ %@LEX%@ %@NumberES%@  [? - %@LEX%@]*"  |
      hfst-regexp2fst > cont2
echo "%@Nouns%@ %@LEX%@ %@Nouns%@  [? - %@LEX%@]*"  |
      hfst-regexp2fst > cont3
hfst-disjunct cont1 cont2 | hfst-disjunct - cont3 |
    hfst-repeat -f 1 > conts
hfst-concatenate start conts |
    hfst-concatenate - end > morphotactics
hfst-compose bag_of_morphs  morphotactics > lexicon
```

Fig. 11. Composing the morphotactics with the disjunction of morphs into a lexicon

Since many morphotactic formalisms tend to be some variation of the item and arrangement scheme, we can build on this to generalize into other notations, since the morphotactics itself does not depend on the description language of the morphotactics. This also makes additions like compound-based weighting of the language-model [13] generally applicable.

For Lexicographers this approach gives the option to choose their favorite notation, e.g. hfst-lexc, Hunspell or Apertium and then continue with other HFST tools. In theory, it would also be possible to use this morphotactic formula to write implementations for any item-and-arrangement morphology, but in prac-

tice it's easier to simply use some high-level scripting language to convert a lexical database into e.g. lexc notation.

3.2 Performance

The HFST toolkit has been implemented using three back-end libraries: SFST, OpenFst and foma. The libraries are linked with HFST. Usually one library is chosen and used throughout the compilation of a given morphology. We can thus compare how different back-end libraries perform in the same task.

In Table 3, we show compilation times for different morphologies using different HFST back-ends. The morphologies are OMorFi [19] for Finnish, Morphisto [31] for German, Morph-it [30] for Italian, Swelex [10] for Swedish and TrMorph [6] for Turkish. OMorFi, Morphisto and TrMorph have several rules for inflection and compounding, Morph-it and Swelex are basically word lists. We use HFST version 3.3.4, with SFST, OpenFst and foma as back-ends. The times are averages from runs of 10 compilations.

Table 3. Compilation times for different morphologies with different HFST back-ends. The times are given in minutes and seconds and averaged over 10 compilations. HFST version is 3.3.4.

Back-End	Finnish	German	Italian	Swedish	Turkish
SFST	2:48	2:12	0:30	0:13	0:12
OpenFst	7:52	7:45	2:24	0:49	0:40
foma	1:52	1:33	0:31	0:13	0:05

In Table 4, we show both the current compilation times and the ones that we achieved in an earlier benchmarking [11] for Finnish and German. We also show the back-end versions used.

Table 4. Compilation times for different morphologies with different HFST back-ends and their versions. The times are given in minutes and seconds.

Back-End	version	Finnish	German
SFST	1.4.2	5:02	6:39
	1.4.6	2:48	2:12
OpenFst	1.2.7	6:51	6:28
	1.2.10	7:52	7:45
foma	0.1.14	1:49	1:29
	0.1.16	1:52	1:33

It can clearly be seen that the performance of HFST with SFST as a back-end has improved: the compilation time of the Finnish morphology has almost

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

halved and the compilation time of the German morphology is only one third of the time at the previous benchmarking. This improvement comes from the newer version of SFST that features more optimized composition and Hopcroft minimization functions. The improved functions were developed by Helmut Schmid in cooperation with the HFST team.

4 Building Taggers

The HFST library includes tools for constructing statistical part-of-speech (POS) taggers which resemble Hidden Markov Models (HMM) from tagged training data. HFST taggers differ from other HMM taggers such as Tnt [4] and Hunpos [8] in that HFST allows combining different estimates for tag probabilities during tagging. It is possible to use e.g. the surrounding word forms in estimating the probability of a given tag. This is more thoroughly explained in [26,27] The accuracy for basic HMM taggers implemented using HFST tools is comparable to the accuracy of Tnt and Hunpos as demonstrated below.

HMM-type taggers include a lexical model and a tag sequence model. The lexical model is needed for determining probabilities for the co-occurrence of tags and word forms disregarding context, and the tag sequence model is used for determining the probabilities for the co-occurrence of tags of neighboring words. The lexical models in HFST taggers include suffix guessers which can be modified to suit the needs of particular languages. Additionally HFST supports using morphological analyzers in tagging.

4.1 The structure of HFST Taggers

HMM taggers are statistical models which determine the most likely POS tag from some tag set for each word in a sentence. For determining the most likely tags, the taggers use lexical probabilities $\mathrm{p}(w|t)$ for each word w and each tag t together with *transition probabilities* for the tag of a word at a given position given the tags of the preceding words $\mathrm{p}(t_i|t_{i-1},\ \ldots t_{i-n})$ [27]. The integer n determines how many preceding tags are considered when estimating the probability of tags. It is called the *order* of the HMM tagger. Second order HMM taggers are the most common in POS tagging.

HFST taggers extend traditional HMM taggers by allowing modifications to the traditional estimate $\mathrm{p}(t_i|t_{i-1},\ \ldots t_{i-n})$. In addition to preceding tags also succeeding tags and word forms can be used when deciding the probability of a tag at a given position. These estimates are combined using finite-state calculus.

In HFST taggers, the lexical probabilities $\mathrm{p}(w|t)$ are given by the lexical component of the tagger and the transition probabilities $\mathrm{p}(t_i|t_{i-1},\ \ldots t_{i-n})$ are given by the sequence component of the tagger. Both models are trained using tagged training data.

Both components can be modified to suit the needs of a particular language. In [27] it is explained how the sequence model can be modified to include preceding and succeeding words in the estimates of the probabilities of tags, and

how suffix guessers can be modified to better suit agglutinative languages. In the present paper we show how a morphological analyzer can be integrated with an HFST tagger to improve the accuracy of the tagger when there are a lot of out-of-vocabulary words.

4.2 The Lexical Model of HFST Taggers

The accuracy of traditional POS taggers suffers greatly because of out of vocabulary (OOV) words, i.e. word forms which were not observed during training of the tagger. OOV words effect the accuracy, since POS taggers generally rely heavily on lexical probabilities $P(word|tag)$ computed for the words occurring in a training corpus.

For languages like English with few productive morphological phenomena, OOV words are not a big problem when there is a lot of training data and the genres of the training data and the data that is tagged are sufficiently similar. When the genres differ considerably there are more OOV words and consequently accuracy is reduced. When building taggers for morphologically rich languages such as Turkish, Finnish or Hungarian, OOV words become a major problem even when no change of genre is involved and even when there is a lot of training data. In agglutinative languages OOV words arise from productive morphological phenomena such as inflection, derivation and compounding.

Usually, e.g. in [4], OOV words are handled using suffix guessers, which combine estimates of all suffixes of the OOV word whose length does not exceed a given threshold. E.g. the POS tag for *ameliorate* can be guessed based on the distribution of tags for the suffixes *-e*, *-te*, ..., *-liorate* if the threshold for the length of suffixes is 7.

HFST taggers offer two improvements for handling of OOV words: (1) the tagger builder can adjust the way in which the probability distribution for different length suffixes are combined to form probability estimates for the OOV word, and (2) the tagger builder has the option to combine taggers with morphological analyzers, whose tag set does not need to equal the tag set of the training data of the tagger.

Adjusting the way suffix estimates are combined is useful e.g. in Finnish, where the suffix guesser proposed by [4] gives poor results. For Finnish, the accuracy of the guesser improves when the only suffix considered is the longest suffix of the word which was seen during training. This is probably a result of the high number of compound words where the final part of the compound is known, but the compound itself is an OOV word. The final part of the compound is generally a very good predictor of the word-class of the compound, so suffixes of the OOV word containing the final part are very informative.

The idea of incorporating a morphological analyzer in a tagger is not new. E.g. [28,18] use morphological analyzers as part of statistical taggers. The novel aspect of HFST taggers is that they can incorporate morphological analyzers whose morphological description differs from the morphological description used in the training data of the statistical tagger. This is useful, because it allows utilizing ready-made linguistic utilities in tagging without the need to address

problems such as differences in the coarseness of the morphological descriptions between the utilities and the training data of the tagger.

In approaches such as [28,18], the tag set of the morphological analyzer and the statistical tagger have to be the same. When an OOV word is encountered, the morphological analyzer is used to look up the possible analyses for the OOV word. These analyses are used directly by the tagger. In contrast, HFST taggers keep track of *ambiguity classes*, which are the sets of morphological analyses emitted by the morphological analyzer for words in the training data. The taggers use the tag distributions of all training data words in a given ambiguity class to determine the tag distribution of an OOV word in the same ambiguity class.

Given an English morphological analyzer, which emits two analyses *dog+N+SG* and *dog+V+INF* for the word form *dog*, the ambiguity class of *dog* is the set of its analyses *{+N+SG, +V+INF}*. In the training data, *dog* might receive Penn Treebank tags like `NN` and `VB`. When the tagger encounters an OOV word in the same ambiguity class as *dog*, such as *man*, it first maps *man* onto its ambiguity class *{+N+SG, +V+INF}* using the morphological analyzer. It then uses the tag distribution of all words in the ambiguity class *{+N+SG, +V+INF}* to estimate the probabilities $P(man|tag)$ for *man* given each tag (e.g. `NN` and `VB`). Note that this does not require that the tag sets of the training data and the morphological analyzer are the same. It also does not require that their morphological descripitions are equally fine-grained.

4.3 Experiments with HFST taggers

We demonstrate HFST taggers by constructing POS taggers for Finnish and English. For English we construct a regular second order HMM tagger. For Finnish we construct two taggers: one regular second order HMM tagger and another second order HMM tagger which utilizes the OMorFi morphological analyzer for Finnish [19].

The English training data and test data come from the Penn Treebank. Sections 1 to 18 were used for training and sections 22 to 24 for testing. The Finnish training and test data come from Finnish newspaper text which has been automatically tagged using the Textmorfo parser[11]. The data is described in Table 5.

Table 5. Data used for training and testing the English and Finnish taggers.

Language	Training data size (tokens)	Test data size (tokens)	Distinct tags
English	912,344	129,654	45
Finnish	1,027,511	156,572	764

[11] http://www.csc.fi/kielipankki/

As Table 6 shows, HFST taggers achieve comparable results to the well known second order HMM tagger TNT. The accuracy for unknown words is slightly worse using the HFST tagger, but for known words the accuracies are nearly identical.

Table 6. Results for second order HMM taggers of English. The accuracy figures for TNT can be found in [8].

English Model	Seen	Unseen	All
TNT	96.77%	85.19%	96.46%
Basic HFST	96.68%	80.71%	96.23%

For Finnish, the accuracy for the basic second order HMM tagger is poorer than for English as seen in Table 7. This is mostly caused by words in the test data that are missing from the training data, i.e. OOV words. In the Finnish test data 11.51% of the words are OOV words. For comparison, only 2.81% of the words in the English test data are OOV words.

To reduce the number of OOV words in the Finnish test data, a morphological analyzer was used as explained above. Using the morphological analyzer, only 2.73% of the words in the test data were OOV words. Consequently a significant increase in total tagging accuracy is seen. See Table 7. The increase is negligible for known words, but significant for unknown words.

Table 7. Results for Finnish taggers. The model Basic HFST is a regular second order HMM tagger. The model With Morph HFST is augmented with a morphological analyzer.

Finnish Model	Seen	Unseen	All
Basic HFST	97.51%	77.51%	95.23%
With Morph HFST	97.53%	83.65%	95.90%

5 Transducer applications

Automata technology is a general framework for describing language models and phenomena in a wide range of linguistic fields from phonology to morphology, as well as certain areas of syntax and semantics (in POS tagging and machine translation). The practical applications cover spell-checking, as demonstrated in Voikko[12] with bindings to LibreOffice, Gnome desktop, Mozilla and Mac OS X

[12] `voikko.sf.net`

Spell Service, to morphological and syntactic analysis as demonstrated in native HFST tools from HFST downloads[13], and to machine translation as demonstrated in several released language pairs in the machine translation system Apertium[14]. This demonstrates a very important feature of HFST, alluded to in previous chapters, i.e. language models can be described with one tool in one theoretical and practical framework. For example, existing morphological analyzers for the Sámi languages found on the Internet[15], written with the Xerox formalism were converted into a spell-checker in one evening and evaluated in [21] with additional training from likewise freely available Sámi Wikipedia. Similar results of using machine translation dictionaries from the free/libre open source project Apertium to create not only dictionaries, but morphological analyzers and spell-checkers, are also demonstrated on our web page.[16]

Regarding the general applicability of finite-state language models in practical applications it may be noted that we can now generate e.g. spell-checkers for a language that has a machine readable dictionary (such as Manx) or a morphological analyzer (such as Greenlandic, which is not easily implemented in other formalisms). The transformation of a dictionary or analyzer in transducer format to a baseline spell-checker (with a homogeneous Damerau-Levenshtein edit distance as an error model) can be performed using finite-state tools without feedback from linguists or native speakers.

5.1 Spellcheckers

The task of finite-state spell-checking is well researched and documented. It consists mainly of two phases, identifying incorrect word forms and creating suggestions for corrections. For incorrect word forms there are two types of mistakes, non-word spelling errors, such as writing *cta* where *cat* is meant, and real-word spelling errors, such as writing *there* where *their* is intended. The method for finding the former in finite-state systems is simply to apply the dictionary to the text word by word. Any unrecognized string not belonging to the language of the dictionary automaton is a non-word spelling error. For real-word errors a statistical n-gram model or syntactic parser is typically required. To correct a spelling error in a finite-state system, a two-tape automaton modeling the typing errors should be applied to the misspelt string to get set of potential corrections [21]. For practical purposes the error model can also be implemented as a fuzzy finite-state traversal algorithm or similar methods [17]. The result of the correction step is a set of word-forms that are correct in the language of the spell-checking dictionary. Another related task is to rank this set to provide the most likely corrections first. A trivial way to perform such ranking would be to use unigram [21] or n-gram probabilities of the words [15] or word-form analyses [22].

[13] `hfst.sf.net`
[14] `www.apertium.org`
[15] `divvun.no`
[16] The examples are available at `http:/www.ling.helsinki.fi/cgi-bin/omor/omordemo.bash`

Creating Spellcheckers. The creation of a finite-state spellchecker involves compiling (at least) two automata: a dictionary that contains the correctly spelled word-forms and the error model that can rewrite misspelt word-forms into correctly spelled ones. The former automaton can be as simple as a reference corpus, containing larger quantities of correctly spelled words and (possibly) smaller quantities of misspelt ones [16]. Also more elaborate dictionaries, such as morphological analyzers described in Section 3 can be used, and also trained for spell-checking purposes with reference corpora without any big modifications to the underlying implementation [21].

For the error-model we can trivially construct an automaton corresponding to the Levenshtein edit distance algorithm [17,23]. For more elaborate error models it is possible to use hunspell algorithms as automata [20] or construct further extensions by hand [21]. Given an aligned error corpus, it is also possible to construct a weighted error model automaton automatically [5].

Checking Strings and Generating Suggestions. String checking is straightforward. It consists of composing the input I with the lexical automaton L and checking whether the output language of the result is empty. If the result is empty, the correction set must be calculated.

A corrected string is a string that can be generated by transforming the input string with the error model and is present in the lexicon. These strings are thus the output language of the composition $I \circ E \circ L$, where E is the error model.

The desired behavior, or result set R, of a spellchecker given input I, a lexicon L and an error model E is thus given by equation 5, where π_2 is the output projection.

$$R = \begin{cases} \pi_2(I \circ L), & \text{if } \pi_2(I \circ L) \neq \emptyset \\ \pi_2(I \circ E \circ L) & \text{otherwise} \end{cases} \tag{5}$$

It is undesirable to compute either of the intermediate compositions $I \circ E$ and $E \circ L$; the former will require futile work (producing strings that are not in the lexicon) and the latter, though possible to precompute, will be very large (see table 8).

Table 8. Size of two lexical transducers composed with Damerau-Levenshtein edit distance transducers, all results minimized

Transducer	states	transitions	SFST file size
Morphalou (French)	77.0K	190.7K	1.7MB
With edit distance 1	9.6K	733.6K	5.7MB
With edit distance 2	18.2K	344.7K	28MB
OMorFi (Finnish)	203.8K	437.9K	4.0M
With edit distance 1	17.7K	16186.6K	120MB
With edit distance 2	30.5K	53738.2K	410MB

To circumvent these problems, a three-way on-line composition of I, E and L was implemented (distributed as `hfst-ospell` under the GPL and Apache licenses). It is along the lines of a more general n-way composition described by Allauzen and Mohri in [1]. The present algorithm is, however, considerably simpler, due to certain implementation details. Firstly, we use an efficient and compact indexing transducer format (`hfst-optimized-lookup`, documented in [25], obviating the need to process the transducers involved into hash tables. Secondly, in the present application it may be guaranteed that no special handling of epsilons is necessary.

We write $I = (Q^I, 0^I, T^I, \Delta^I)$, where Q is the set of states, 0 is the starting state, T is the set of terminal states and Δ is the set of transitions (triples of (state, symbol pair (input and output), state)), and similarity for E and L.

Eliding for the time being weights and flag diacritics, states in the composition transducer are triples of states of the component transducers. Analogously with two-way composition, the starting state is $(0^I, 0^E, 0^L)$ and the edge $\delta = (q_1, (\sigma_1, \sigma_2), q_2)$ exists if edges $(q_1^I, (\sigma_1, \sigma'), q_2^I)$, $(q_1^E, (\sigma', \sigma''), q_2^E)$ and $(q_1^L, (\sigma'', \sigma_2), q_2^L)$ exist in the respective transducers for some $\sigma' \in \Sigma^I \cap \Sigma^E, \sigma'' \in \Sigma^E \cap \Sigma^L$, where $q_n = (q_n^I, q_n^E, q_n^L)$ and $\sigma_1 \neq \epsilon \neq \sigma_2$.

The set of the states of the composition, Q, are the reachable subset (in the sense of having a path from 0 to the state) of $Q^I \times Q^E \times Q^L$.

If in a given state any of the component transducers has an edge involving epsilon and the other symbols match as above, the resulting composed edges go to states such that any state in a transducer to the left of an input epsilon or to the right of an output epsilon is unchanged, eg. if we have $(q_1^I, (\sigma, \sigma'), q^{I'})$, $(q_1^E, (\epsilon, \sigma''), q_2^E)$ and $(q_1^L, (\sigma'', \sigma'''), q_2^L)$, a successor state of q_1 will be (q_1^I, q_2^E, q_2^L).

There is no special handling of cases where an epsilon output and an epsilon input occur simultaneously in consecutively acting transducers. The algorithm is however guaranteed to terminate in the present application as long as epsilon cycles do not occur on the input side of the transducers and I is acyclic. That being the case, the state reached in I is a loop variant that increases (towards termination) whenever E consumes input. For this not to happen for an indefinitely large number of iterations, either E itself would have to have an input epsilon cycle, or indefinitely many states in the composition would have to be created such that the state of E is unchanged. This would require L to have an input epsilon cycle.

The final states are the reachable subset of $T = T^I \times T^E \times T^L$.

It can trivially be verified that this is equivalent to taking the two compositions $(I \circ E) \circ L$.

This three-way composition is computed in a breadth-first manner with a double-ended queue of states. The queue is initialized with the starting state, and the target of every edge is computed and pushed onto the queue. The starting state is then discarded, and the process is repeated with a new state popped from the queue until the queue is empty.

Conceptually, the state space of $E \circ L$, which contains all the misspellings (in the sense of E) of all the entries in the lexicon, is explored in such a way that only the states visited when looking up I are generated.

For this process to be guaranteed to terminate, it is sufficient that none of the component transducers have input-epsilon loops and that the input transducer accepts strings of only finite length. This is because every newly generated state will either have a shorter sequence of edges to traverse in I ("increment q^I") or be closer to requiring an edge in I to be traversed (due to a finite sequence of epsilon edges becoming shorter), establishing a loop variant.

Weights representing the probability of a particular correction being the correct one are a natural extension, and in `hfst-ospell` correspond to multiplication in the tropical semiring (for details see [2]) of the weight each edge traversed. Multiplication in this semiring is the standard addition operation of positive real numbers, which we approximate by addition of `float`s. Each state in the queue is recorded with an accumulated weight, and its successor states have this weight incremented by the sum of the weights of the edges traversed in the component transducers.

The alphabet of I cannot be determined in advance, and in practice is taken to be the set of Unicode code points. To allow the error model to correct unexpected symbols in this large space, `hfst-ospell` uses a special symbol, `@_UNKNOWN_SYMBOL_@` which is taken to be equal to any symbol that is otherwise absent.

Error model tool and optimizations. For the most common case of generating Levenshtein and Damerau-Levenshtein (in which transposition of adjacent symbols constitutes one operation) distance error models, a tool (`editdist.py`) and definition format was developed.

The definition format serves the purposes of minimizing the number of symbols used in the error model (the number of transitions is $O(|\Sigma|^2)$) and introducing weights for edits. Typical morphologies have a number of unusual characters (punctuation, special symbols) or internally used symbols (eg. flag diacritics) that should be filtered for more efficient correction. This is accomplished by providing a facility for reading the alphabet from a transducer, ignoring any symbols of more than one Unicode code point, and reading further ignorable symbols from a configuration file.

The configuration file allows specifying weights to be added to any edit involving a particular character or a particular edit operation (for example, assigning a low weight to the edit o → ö for an OCR application).

Certain characteristics of the correction task permit efficiency-oriented improvements to error models. A naive Levenshtein error model with edit distance 2 in `ospell` would, when given the word word `arrivew` where the French word `arriver` was intended, do the following useless work, where e.g. a:0 means output epsilon for input a:

```
a:0 0:a r r i v e    [failure]
0:a a:0 r r i v e    [failure]
```

```
a r:0 0:r r i v e    [failure]
a 0:r r:0 r i v e    [failure]
...
a r r i v e w:0      [success]
a r r i v e w:z      [success]
a r r i v e w:r      [success]
...
```

When the correctable error is near the end, almost every symbol is deleted and inserted with no effect. This may be circumvented by adding, for each deletion and insertion, a special successor state from which its inverse is absent.

Ranking Suggestions. When applying the error model and the language model to input with spelling errors, the result is typically an ordered set of corrected strings with some probability associated with each correction. After applying the contextless error correction described earlier, it is possible to use context words and their potential analyses to re-rank the corrections [22].

6 Discussion and Future Work

6.1 Synonym and Translation Dictionaries

Other finite-state applications created with HFST include inflecting thesauri and translation dictionaries. These applications have been created from the bilingual parallel Princeton WordNet and FinnWordNet. The creation of FinnWordNet is documented in [12]. FinnWordNet contains roughly 150,000 word meanings in Finnish with their English translations. The synonym dictionaries for Finnish and English and the Finnish-English and English-Finnish translation dictionaries as well as their demos can be found on `hfst.sf.net`.

The inflecting synonym dictionaries were created as a composition of three transducers: (1) a morphological analyzer, (2) a transducer that replaces one word with another while copying the inflection tags and (3) a morphological generator as an inversion of the morphological analyzer. The translation dictionaries only have components (1) and (2). In the future, we intend to take advantage of the weighted transducers to introduce contexts so as to be able to suggest the most likely synonym or the most likely translation in context.

6.2 Extending Transducers for Pattern Matching

Advanced, fast and flexible pattern matching is a major requirement for a variety of tasks in information extraction, such as named entity recognition. An approach to this task was presented by Lauri Karttunen in [9], and an outline for implementing it in HFST is given here.

Some desiderata for pattern matching. Several patterns, including nested matches, should be able to operate in parallel, and it should be possible to impose contextual requirements (rules) on the patterns to be matched. Matching should be efficient in space and time — in particular, it should be possible to avoid long-range dependencies which are awkward for FST transformations. A powerful system should also have a facility for referring to common subpatterns by name.

The pmatch Approach. For a more detailed overview the reader is directed to [9]. Here we focus on the aspects of `pmatch` that necessitate extensions to a FST formalism from the point of view of the implementation.

`pmatch` is presented as a tool for general-purpose pattern matching, tokenizing and parsing. It is given a series of definitions and a specialized regular expression, and it then operates on strings, passing through unmodified any parts of them that fail to match the expression, and applying transformations to any matched parts. If several matches beginning from a common point in the input can be made, matches of less than maximal length are considered invalid.

The expressions may refer to other expressions (possibly combined with each other by operations on regular languages), contextual conditions (left or right side, with negation, `OR` and `AND`) and certain built-in transformation rules. The most interesting of these transformation rules is `EndTag()`, which triggers a wrap-around XML tag to be inserted on either side of the match.

Referencing other regexes (including self-reference) is unrestricted, so the complete system has the power of a recursive transition network (RTN), and matching is therefore context-free.

The crucial extension-demanding features for an HFST utility with similar applications are:

- A distinction between an augmented universal transducer (the top level which echoes all input in the absence of matches) and sub-transducers
- Ignoring non-maximal matching, ie. a left-to-right longest-distance matcher
- An unrestricted system for referencing other transducers by name
- Special handling of `EndTag()` and instructions for context checking
- Reserved symbols for implementing transducer insertion/reference, `EndTag()` and contexts

The referencing system is apparently the only one of these that would be impossible to implement in a strict FST framework; the other extensions suggest compilations to larger, possibly less efficient transducers.

7 Conclusion

HFST—Helsinki Finite-State Technology (`hfst.sf.net`) is a framework for compiling and applying linguistic descriptions with finite-state methods. We have

demonstrated how HFST uses finite-state techniques for creating runtime morphologies, taggers, spellcheckers, inflecting synonym dictionaries as well as translation dictionaries using one open-source platform which supports extending the descriptions with statistical information to allow the applications to take advantage of context. HFST offers a path from language descriptions to efficient language applications.

Acknowledgments

We wish to acknowledge the FIN-CLARIN and META-NORD projects for their financial support as well as HFST users for their many constructive suggestions. Additonally, the research leading to some of these results has received funding from the European Commission's 7th Framework Program under grant agreement n° 238405 (CLARA). Miikka Silfverberg's work is funded by LANGNET.

References

1. Allauzen, C., Mohri, M.: N-way composition of weighted finite-state transducers. International Journal of Foundations of Computer Science 20, 613–627 (2009)
2. Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., , Mohri, M.: OpenFst: A general and efficient weighted finite-state transducer library. In: Proceedings of the 12th International Conference on Implementation and Application of Automata, CIAA 2007. pp. 11–23 (2007), `http://www.openfst.org`
3. Beesley, K.R., Karttunen, L.: Finite State Morphology. CSLI publications (2003)
4. Brants, T.: Tnt - a statistical part-of-speech tagger. In: Proceedings of the Sixth Applied Natural Language Processing (ANLP-2000). Seattle, WA (2000)
5. Brill, E., Moore, R.C.: An improved error model for noisy channel spelling correction. In: ACL '00: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics. pp. 286–293. Association for Computational Linguistics, Morristown, NJ, USA (2000)
6. Çöltekin, Ç.: A freely available morphological analyzer for Turkish. In: Proceedings of the 7th International Conference on Language Resources and Evaluation (LREC2010) (2010)
7. Gerdemann, D., van Noord, G.: Transducers from re-write rules with backreferences. In: Proceedings of the EACL Conference. pp. 126–133 (1999)
8. Halácsy, P., Kornai, A., Oravecz, C.: Hunpos—an open source trigram tagger. In: ACL-2007. Prague, Czech Republic (2007)
9. Karttunen, L.: Beyond morphology: Pattern matching with fst. In: Mahlow, C., Piotrowski, M. (eds.) Systems and Frameworks for Computational Morphology. Communications in Computer and Information Science, vol. 100, pp. 1–13. Springer, Berlin Heidelberg (2011)
10. Kempe, A., Karttunen, L.: Parallel replacement in finite state calculus. In: In the Proceedings of the 16th International Conference on Computational Linguistics. pp. 622–627 (1996)
11. Lindén, K., Axelson, E., Hardwick, S., Silfverberg, M., Pirinen, T.: Hfst—framework for compiling and applying morphologies. In: Mahlow, C., Pietrowski, M. (eds.) State of the Art in Computational Morphology. Communications in Computer and Information Science, vol. 100, pp. 67–85. Springer Berlin Heidelberg (2011)

12. Lindén, K., Carlson, L.: Finnwordnet—wordnet på finska via översättning. LexicoNordica 17 (2010)
13. Lindén, K., Pirinen, T.: Weighting finite-state morphological analyzers using hfst tools. In: FSMNLP 2009 (2009)
14. Lindén, K., Silfverberg, M., Pirinen, T.: Hfst tools for morphology—an efficient open-source package for construction of morphological analyzers. In: Mahlow, C., Pietrowski, M. (eds.) State of the Art in Computational Morphology. Communications in Computer and Information Science, vol. 41, pp. 28–47. Springer Berlin Heidelberg (2009)
15. Mays, E., Damerau, F.J., Mercer, R.L.: Context based spelling correction. Inf. Process. Manage. 27(5), 517–522 (1991)
16. Norvig, P.: How to write a spelling corrector. Web Page, Visited February 28th 2010, Available `http://norvig.com/spell-correct.html` (2010)
17. Oflazer, K.: Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. Computational Linguistics 22(1), 73–89 (1996)
18. Oravecz, C., Dienes, P.: Efficient stochastic part-of-speech tagging for Hungarian. In: Proceedings of the Third International Conference on Language Resources and Evaluation. pp. 710–717. Las Palmas (2002)
19. Pirinen, T.: Suomen kielen äärellistilainen automaattinen morfologinen analyysi avoimen lähdekoodin menetelmin. Master's thesis, Helsingin yliopisto (2008), `http://www.helsinki.fi/~tapirine/gradu/`
20. Pirinen, T.A., Lindén, K.: Creating and weighting hunspell dictionaries as finite-state automata. Investigationes Linguisticae 19 (2010)
21. Pirinen, T.A., Lindén, K.: 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. pp. 13–18. Valletta, Malta (2010)
22. Pirinen, T.A., Silfverberg, M., Lindén, K.: Context-sensitive spelling correction. In: Gelbukh, A. (ed.) International Conference on Intelligent Text Processing and Computational Linguistics. New Delhi, India (2012)
23. Savary, A.: Typographical nearest-neighbor search in a finite-state lexicon and its application to spelling correction. In: CIAA '01: Revised Papers from the 6th International Conference on Implementation and Application of Automata. pp. 251–260. Springer-Verlag, London, UK (2002)
24. Silfverberg, M., Lindén, K.: Conflict resolution using weighted rules in hfst-twolc. In: Proceedings of the 17th Nordic Conference of Computational Linguistics NODALIDA 2009. pp. 174–181. Nealt (2009)
25. Silfverberg, M., Lindén, K.: Hfst runtime format—a compacted transducer format allowing for fast lookup. In: Watson, B., Courie, D., Cleophas, L., Rautenbach, P. (eds.) FSMNLP 2009 (13 July 2009), `http://www.ling.helsinki.fi/~klinden/pubs/fsmnlp2009runtime.pdf`
26. Silfverberg, M., Lindén, K.: Part-of-speech tagging using parallel weighted finite-state transducers. In: Proceedings of the 7th International Conference on NLP, IceTAL 2010 (2010)
27. Silfverberg, M., Lindén, K.: Combining statistical models for POS tagging using finite-state calculus. In: Proceedings of the 18th Conference on Computational Linguistics, NODALIDA 2011. pp. 183–190 (2011)
28. Tzoukermann, E., Radev, D.: Using word class for part-of-speech disambiguation. In: Proceedings, Fourth Workshop on Very Large Corpora WVLC'96. Copenhagen, Denmark (1996)

29. Yli-Jyrä, A.: Transducers from parallel replace rules and modes with generalized lenient composition. In: Finite-state methods and natural language processing (2008), http://www.ling.helsinki.fi/users/aylijyra/all/YliJyra-2008b:trafropar:inp.pdf
30. Zanchetta, E., Baroni, M.: Morph-it! a free corpus-based morphological resource for the Italian language. Corpus Linguistics 2005 1(1) (2005)
31. Zielinski, A., Simon, C.: Morphisto – an open source morphological analyzer for German. In: Proceeding of the 2009 conference on Finite-State Methods and Natural Language Processing: Post-proceedings of the 7th International Workshop FSMNLP 2008. pp. 224–231. IOS Press Amsterdam, The Netherlands (2009)