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

Krister Lindén Miikka Silfverberg
Tommi Pirinen
University of Helsinki
firstname.lastname@helsinki.fi

March 6, 2010

Abstract

Morphological analysis of a wide range of languages can be implemented efficiently using finite-state transducer technologies. Over the last 30 years, a number of attempts have been made to create tools for computational morphologies. The two main competing approaches have been parallel vs. cascaded rule application. The parallel rule application was originally introduced by Koskenniemi [1983] and implemented in tools like TwolC and LexC. Currently many applications of morphologies could use dictionaries encoding the a priori likelihoods of words and expressions as well as the likelihood of relations to other representations or languages. We have made the choice to create open-source tools and language descriptions in order to let as many as possible participate in the effort. The current article presents some of the main tools that we have created such as HFST-LexC, HFST-TwolC and HFST-Compose-Intersect. We evaluate their efficiency in comparison to some similar tools and libraries. In particular, we evaluate them using several full-fledged morphological descriptions. Our tools compare well with similar open source tools, even if we still have some challenges ahead before we can catch up with the commercial tools. We demonstrate that for various reasons a parallel rule approach still seems to be more efficient than a cascaded rule approach when developing finite-state morphologies.

1 Introduction

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

*This is author's draft; it may differ from print version, especially since hyperref and llncs style don't work together. This article was published in Proceedings of SFCM 2009

Over the last 30 years, a number of attempts have been made to create tools for computational morphologies and some of them have withstood the test of time better than others. A major effort that has shaped the landscape and incorporated many lasting ideas is the morphological development tools created by Xerox. It started with the insight that we can use transducers to describe or encode phonological processes and relate various levels of linguistic abstraction using tools like TwolC introduced by Koskenniemi and Karttunen [1983, 1987, 1992]. To efficiently compile large-scale lexicons into transducers, we need special lexicon compilers like LexC described by Karttunen [1993, 1994].

Such tools do not solve all the problems. Writing full-scale dictionaries in LexC may well be compared to having programmers write sophisticated applications in C without access to any of the modern high-level libraries. It is possible, but unless it is done in some principled way, one may easily end up with spaghetti-code that is difficult to maintain. This is not the fault of the lexicon compiler, but the general programming solution is to create several descriptions that are small and independent, i.e. modular. With this insight and as computers became more powerful, the initial calculus that was conceived for abstract objects like automata and transducers in TwolC and LexC was expanded and migrated into the lexical programming environment xfst documented by Beesly and Karttunen [2003], where smaller lexical modules for various purposes can be tailored and combined using finite-state calculus operations.

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

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

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

Recently, a number of open-source finite-state processing environments have emerged, e.g. for unweighted transducers there are the *SFST–Stuttgart Finite-State Transducer Tools*[2] by Schmid [2005], *foma: a finite-state machine toolkit*

[1] `https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstRuntimeInterface`

[2] `http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html`

and library[3] by Huldén, etc., and for weighted transducers there are *Vaucanson*[4] by Lombardy et al. [2004], *OpenFST Library*[5] by Allauzen et al. [2007], etc. These are valuable contributions to the open source software that we can build on.

Our particular goal currently is in providing the basic facilities for efficiently developing, compiling and running morphologies with or without weights. To achieve our goal, we decided to create a unified API[6], which is capable of interfacing various weighted and unweighted finite-state transducer libraries allowing us to incorporate new libraries as needed. Currently, we have interfaces to SFST and OpenFST. On top of the unified API, we created a set of basic tools[7], e.g. HFST-TwolC, HFST-LexC, HFST-Compose-Intersect, HFST-Test, HFST-Lookup, etc. With these tools, we created or used real full-fledged morphological descriptions of different languages from different language-families[8], e.g. *English, Finnish, French, Northern Sámi* and *Swedish*. We used the morphological descriptions for testing the functionality of the tools and for evaluating the performance of the different libraries through a unified interface on the full-fledged morphological development and compilation tasks.

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

2 HFST-LexC

A lexicon compiler is a program that reads sets of morphemes and their morphotactic combinations in order to create a finite-state transducer of a lexicon. This finite state transducer is called a *lexical transducer*[1994]. The lexical transducer may be further adjusted with e.g. phonological rules. The example for our lexicon compiler is set by LexC of Xerox [2003]. In LexC, morphemes are arranged into named sets called sub-lexicons. Each entry of a sub-lexicon is a pair of finite strings[9] associated with the name of a sub-lexicon called a *continuation class*.

Below, we highlight the main design decisions that influenced the efficiency of

[3] `http://foma.sourceforge.net/`

[4] `http://www.lrde.epita.fr/cgi-bin/twiki/view/Projects/Vaucanson`

[5] `http://www.openfst.org/`

[6] `http://www.ling.helsinki.fi/kieliteknologia/tutkimus/hfst/documentation.shtml`

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

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

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

the implementation and some of the we present the main theory for compiling a LexC description into a finite-state transducer. Our morphology example outlines the nominal inflection of four Finnish nouns as shown in Table 1. This example is a highly simplified version of the actual morphology.

```
Multichar_Symbols

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

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

LEXICON N1b
    NounSg ;
    NounPtvA ;


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

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

LEXICON Compounding
    Root ;

LEXICON Ennd
    # ;
```

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

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

2.1 Efficient Tokenizing of a Sub-Lexicon Entry

Lexicon entries are tokenized using a simple left-to-right longest match algorithm. The entry can be tokenized by going through the entry string, position by position, and looking up the longest symbols available using a very simple greedy algorithm. An alternative, but less efficient, strategy is to compose the entry string with a tokenizer-transducer, implementing greedy left-to-right matching, to achieve the correct partition.

2.2 Efficient Union of Sub-Lexicon Entries

The finite-state form of a sub-lexicon is a union of entry transducers. Building a union of entry transducers is a relatively straight-forward process. However, iteratively taking the union of n entries with the n+1th entry is not ideal. A faster approach, given that all our entries are simple finite strings is to build the sub-lexicon transducer as a prefix tree, *trie*.

2.3 Efficient Implementation of Morphotax over Sublexicons

There are two strategies for making the sub-lexicon combinations. A trivial, and fast, strategy is to connect the entries using continuation classes to sublexicons of the same name with an epsilon arc. An alternative method, which we implemented, is to use named auxiliary symbols and the finite-state algebra to create overgenerating combinations which are filtered by composition to achieve legal combinations. This is further described in the next subsection.

2.3.1 Combining sublexicons using standard finite-state transducer algebra.

We assume all standard finite state operations to be known. For an introduction, see Beesly and Karttunen [2003]. We use the following notation: $\cup$ is union, $\cap$ is intersection, $\circ$ is composition, juxtaposition is concatenation. Latin characters represent symbols of language and the ε symbol is used for a zero-length string. Capital Greek letters Σ, Γ represent subsets of an alphabet. We define $\Sigma = \{a, b, \ldots\}$ as a subset of the alphabet used for representing the morphophonology of the language in LexC definitions. Γ is the alphabet of the *auxiliary* symbols used in our rules in the morphotax implementation. We assume that $\Sigma \cup \Gamma = \emptyset$. We use the symbol $J \in \Sigma$ for *joiners* to delimit and combine morphemes in our morphotax. A joiner for an entry with a continuation class named x is denoted as J_x and a joiner for a sub-lexicon named y is denoted as J_y.

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

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

$$\mathcal{M} = \Sigma^\star \; J \tag{1}$$

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

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

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

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

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

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

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

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

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

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

E.g., for the sublexicons *Root*, *N1b*, *NounSg*, and *Ennd* in Table 1, and their entries *akku* and *+sg+all:lle*, we get the disjunction of lexicons $L^\star$, which we filter using $L^\star \circ F'$ as shown in Table 2.

$L^\star =$	($J_{Root}\ a\ k\ k\ u\ J_{N1b}$ | $J_{N1b}\ J_{NounSg}$ |
	$J_{NounSg}\ {+}sg{:}l\ {+}all{:}l\ \varepsilon{:}e\ J_{Ennd}$ | $J_{Ennd}\ J_{\#}$)*
$F =$	$J_{Root}\ J_{Root}$ | $J_{N1b}\ J_{N1b}$ | $J_{NounSg}\ J_{NounSg}$ | $J_{Ennd}\ J_{Ennd}$
$L^\star \circ F' =$	$J_{Root}\ a\ k\ k\ u\ J_{N1b}\ \ J_{N1b}\ J_{NounSg}$
	$J_{NounSg}\ {+}sg{:}l\ {+}all{:}l\ \varepsilon{:}e\ J_{Ennd}\ \ J_{Ennd}\ J_{\#}$

Table 2: Filtering a single path in HFST-LexC with a morphotax filter.

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

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

3 HFST-TwolC

Two-level rules are parallel constraints on symbol-pair strings governing the realizations of lexical word-forms as corresponding surface-strings. They were introduced by Koskenniemi [1983] and have been used for modeling the phonology of numerous natural languages. HFST-TwolC is an accurate and efficient

open-source two-level rule compiler. It compiles grammars of two-level rules into sets of finite-state transducers. The rules are represented as regular-expression operations closely resembling familiar phonological re-write rules both to appearance and semantics.

The most widely known two-level rule-compiler existing at the moment is the *Xerox Two-Level Rule Compiler* (later TwolC) presented by Karttunen el al. [1992]. It is proprietary software, which imposes some limitations upon its use. The HFST-TwolC compiler has been designed to be an open-source substitute for the TwolC compiler and has a syntax and semantics very similar to those of the TwolC compiler. Hence existing two-level grammars, designed to compile under the TwolC compiler, require very few modifications to compile correctly under HFST-TwolC.

Besides being an open-source program, HFST-TwolC also has other benefits compared with the TwolC compiler. Resolution of rule-conflicts is an important part of compiling two-level grammars. We know of at least one instance, where the TwolC compiler resolves rule-conflicts in an incorrect way (see Sect. 3.2.2). It also compiles epenthesis rules in a way, which denies the grammar-writer the full expressive power of two-level rules (see Sect. 3.2.1). In HFST-TwolC we have been able to remedy these shortcomings by compiling the rules with the *generalized restriction-operation* (later *GR-operation*), presented by Yli-Jyrä and Koskenniemi [2006]. It allows compilation of two-level rules in a uniform way and makes conflict-resolution easy to tackle, while still permitting efficient compilation.

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

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

We use the GR-operation to compile the grammar-rules in HFST-TwolC. In Sect. 3.2.1, we explain how the different types of two-level rules are compiled. Rule-conflicts and their resolution are covered in Sect. 3.2.2.

3.1 An Example Grammar

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

3.1.1 The Alphabet.

The alphabet of a two-level grammar contains all lexical symbols specified in the HFST-LexC grammar together with their possible surface realizations. In the example grammar in Table 3, the alphabet contains all letters used in

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

```
Alphabet

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

Rule-variables

   Cm Cs Cw ;

Sets

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

   Vowels = UpperCaseVowels LowerCaseVowels ;

Definitions

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

Rules

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

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

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

Table 3: An example HFST-TwolC grammar governing the surface realizations of the forms presented in the example lexicon in Table 1.

Finnish words together with the vowel-harmony archphoneme `~A`, the gradation morphophonemes `~K` and `~P`, as well as, the gradation-markers `+AV+`, `+AV-`, `+AVA`, `+AVD`, `+AVH`, `+AVM`.

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

The letters in the example-grammar of Table 3 always correspond to themselves on the surface. The gradation-markers always correspond to zero and the archphoneme `~A` and the morphophonemes `~K` and `~P` have various surface-realizations. E.g. `~A` is always realized as either `a` or `ä`.

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

Declaring all symbol-pairs in HFST-TwolC is mandatory, as we have not yet implemented an *other-symbol* like the one in Xerox TwolC [1992] using the HFST interface. Besides the grammar-formalism, this also affects the compile-time for rules, which becomes more dependent on the number of symbol-pairs in the grammar.

3.1.2 The Rule-variables.

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

3.1.3 The Sets.

It is often convenient to name some classes of symbols, which are used in many rules. E.g. the class `BackVowels` in the example-grammar in Table 3, which contains all vowel-segments used in the grammar. The sets in HFST-TwolC and TwolC are very similar constructs.

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

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

3.1.4 The Definitions.

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

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

Table 4: Symbol-pair correspondences for demonstrating the vowel-harmony rules.

3.1.5 The Rules.

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

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

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

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

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

The rule accepts the first correspondence in Table 4 since the vowel preceding `~A` is `y`, which is not a back-vowel. It disallows both of the latter correspondences. In the second correspondence `~A` is realized as `a`, even though the preceding surface-vowel is not a back-vowel. This violates the `=>` direction of the rule. In the third correspondence, `~A` is realized as `ä`, but the preceding surface-vowel is `u`, which is a back-vowel. This violates the `<=` direction of the rule.

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

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

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

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

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

3.2 Compiling the Rules and Resolving Rule-Conflicts

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

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

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

We use the GR-operation of Yli-Jyrä and Koskenniemi [2006] to compile rules. Both compiling rules and conflict-resolution is simplified using the operation.

The compilation in HFST-TwolC differs from TwolC when epenthesis rules are compiled. As Yli-Jyrä and Koskenniemi [2006] point out, epenthesis rules may be compiled as any other surface-requirement rules using the GR-operation. This increases the expressive power of the two-level grammar as explained below.

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

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

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

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

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

3.2.1 Compiling one rule.

Yli-Jyrä and Koskenniemi [2006] explain how ordinary two-level rules can be compiled using the GR-operation. We use slight variations of the same methods.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2.2 Resolving rule-conflicts.

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

conflict resolution of the XEROX compiler sometimes works in an unexpected way (see example below).

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

Consider the rules

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

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

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

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

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

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

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

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

4 HFST-Compose-Intersect

A lexicon compiled using HFST-LEXC and a grammar of two-level rules compiled using HFST-TWOLC are combined using the program HFST-COMPOSE-INTERSECT. It is an implementation of the *intersecting composition* algorithm presented by Karttunen [1994]. The result of the operation is equivalent to the composition of the lexicon-transducer with the intersection of the rule-transducers.

Karttunen [1994] observed that the intersection of the rule-transducers alone may be extremely large and computing it may take a long time, whereas intersecting composition allows the lexicon to restrict the intersection of the rule-transducers. This reduces compilation time significantly.

Although computers have become considerably faster since 1994 and they have more memory, computing the intersection of the rule-transducers can still be very space-consuming. We intersected the rule-transducers of the two-level

implementation of OMorFi[12], i.e. Pirinen's [2008] morphological analyzer for Finnish. Without the intersecting composition, the rule intersection took eleven hours using the same machine we used for conducting our other performance-tests. Hence we believe that intersecting composition is still a necessary operation when developing full-scale two-level morphological analyzers.

5 Full-Scale Morphological Analyzers using HFST Morphological Tools

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

The French analyzer was built from the existing morphological full-form lexicon Morphalou[13]. The lexicon was translated into the LexC format and it contains some 550,000 entries in a single lexicon. Each entry represents a word form and its analysis. We chose this lexicon for testing HFST-LexC with a large number of real entries.

The Finnish analyzer has two implementations, i.e. the version using the SFST compiler format of OMorFi which is Pirinen's [2008] original analyzer for Finnish, and a reformulated version using a LexC lexicon and a TwolC grammar format. The reformulation was done manually by converting the morpheme sets of the original code into LexC sublexicons and rewriting the phonological rules from replace cascades into TwolC rule sets. While care has been taken to ensure the similarity of the implementations, it should be noted that the versions are not totally equivalent. We still think they are close enough for a meaningful comparison of the two approaches.

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

The characteristics of the analyzers of the three languages are summarized in Table 5. The first three of the columns summarize the HFST-LexC lexicons stating the numbers of sublexicons, lexicon-entries and symbols used in the lexicons. The remaining three columns summarize the HFST-TwolC grammars. They state the numbers of symbol-pairs, rules and subrules in the double-sided rules and rules with variables. The example for French has no entries in the last three columns, since it has no two-level grammar.

6 Performance Evaluation

The goal of the performance evaluation is is to see how far we still have to go before we reach industrial-strength performance. Additionally, we wish to

[12] `http://kitwiki.csc.fi/kitwiki/Main/OMorFiHome`
[13] `http://www.cnrtl.fr/lexiques/morphalou/`
[14] `http://www.divvun.no`

	HFST-LexC			HFST-TwolC		
Language	Sublexicons	Entries	Symbols	Pairs	Rules	Subrules
French	1	553,158	87	—	—	—
Finnish	213	94,278	301	169	12	76
Northern Sámi	870	105,503	428	313	105	555

Table 5: Some numbers characterizing the lexicons and two-level grammars we used for testing.

see how the performance of the LexC and TwolC approach with parallel-rules compares to the approach with cascaded-rules. To achieve these goals, we compare HFST with some other open source tools and an industrial strength implementation by Xerox. By compiling the analyzers mentioned in the previous section, we can also collect performance figures on real full-fledged morphologies for identifying the most significant remaining bottle-necks in our tools.

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

As the Finnish OMorFi analyzer has two almost identical versions: one using replace-rules for the SFST compiler and one using two-level rules for the LexC and TwolC tools, we are able to compare the efficiency of the two approaches to building morphological analyzers.

Below, we have five tables summarizing the results of the performance tests. The first Table 6 compares the total compile times of the analyzers using the HFST tools, the Xerox tools, the foma LexC tool and the SFST compiler. For foma, only the compile-time for the analyzer of French is given, as foma only comes with a LexC[15] interface. For the SFST compiler, only the compile-time for the Finnish lexicon is given, as our only implementation with cascaded rules is for Finnish.

Language	HFST tools	foma LexC	SFST compiler	Xerox tools
French	45.92 s	16.87 s	—	5.46 s
Finnish	25.42 s	—	1682.04 s	1.83 s
Northern Sámi	287.21 s	—	—	24.61 s

Table 6: Total compile-times using HFST tools, foma LexC, SFST compiler and Xerox tools to compile lexical transducers. Times are in seconds.

The following three Tables 7, 8 and 9 give the HFST compile-times for the analyzers of Finnish, French and Northern Sámi. The times have been broken down into sub-phases of the compilation in order to see where the bottle-necks are. The phases are explained below the tables.

[15] foma also has an xfst interface.

Language	1	2	3	4	Total
French	19.27 s	1.18 s	0.08 s	25.40 s	45.92 s
Finnish	3.59 s	0.19 s	0.29 s	17.99 s	22.05 s
Northern Sámi	3.74 s	0.23 s	2.00 s	78.84 s	84.81 s

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

Table 7: HFST-LexC performance broken into the different phases of the compilation process. Times are in seconds.

Language	1	2	3	Total
Finnish	0.10 s	0.04 s	1.27 s	1.41 s
Northern Sámi	2.11 s	1.35 s	24.77 s	28.23 s

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

Table 8: HFST-TwolC performance broken into the different phases of the compilation process. Times are in seconds.

Finally, Table 10 gives an indication of the maximal memory consumption during the lexicon compilations using the HFST tools and the Xerox tools.

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

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

Language	1	2	3	4	Total
Finnish	0.10 s	1.44 s	0.36 s	0.07 s	1.97 s
Northern Sámi	0.90 s	154.60 s	18.26 s	0.41 s	174.17 s

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

Table 9: HFST-Compose-Intersect performance broken down into the different phases of the compilation process. Times are in seconds.

Language	HFST-LexC	HFST-TwolC	HFST-Compose-Intersect
French	596 MB	—	—
Finish	181 MB	13 MB	48 MB
Northern Sámi	180 MB	291 MB	1090 MB (1.1 GB)
Language	Xerox LexC	Xerox TwolC	
French	85 MB	—	—
Finnish	28 MB	3 MB	—
Northern Sámi	13 MB	12 MB	—

Table 10: Maximum space required using HFST and Xerox utilities to compile the transducers. Space indications are in megabytes (MB).

7 Discussion and Future Research

In this section we discuss the evaluation results and suggest some further lines of research and development of the tools that the evaluation figures seem to indicate. Comparing the total compilation times for HFST tools, Xerox tools, foma LexC and SFST compiler, shows that HFST is still a magnitude slower than the Xerox tools. However, HFST compares well with the other open-source tools. The decrease in compile-time for the Finnish lexicon, when parallel-rules are used, is considerable by improving performance with almost two magnitudes. Most importantly, using the HFST tools is sufficiently quick not to slow down the development of full-scale morphological analyzers. Even large morphological analyzers like the analyzers for French and Northern Sámi already compile in less than ten minutes.

7.1 HFST-LexC Performance

Comparing the HFST-LexC compilation times for French and Northern Sámi given in Table 7, we see that the entry parsing is almost linear. The lexicon for French has about five times as many entries as the lexicon for Northern Sámi. This is a result of the trie-union described in Sect. 2, which speeds up

the building of sublexicons.

We also see that the number of sublexicons is currently very influential on the HFST-LexC compile-time. The lexicon for French only has one sub-lexicon, whereas the lexicon for Northern Sámi has 870 sub-lexicons. This indicates that the current implementation of the morphotax filtering still slows down the compilation with some unnecessary copying.

There are two main parts that dominate the time consumption of the fourth column of Table 7, i.e. alphabet discovery and final determinization and minimization. Since the HFST API and underlying libraries need the full alphabet to be known prior to the compilation of the entries and the LexC file format provides no declaration of the alphabet, HFST-LexC needs to make an initial pass through the entries before the actual compilation phase to gather the alphabet. In addition, the determinization and minimization of the final result consumes well over half of the compile-time of the Northern Sámi lexicon (46,85 seconds), which we estimate is caused by a large number of lexicons containing epsilon entries giving rise to indeterminism.

7.2 HFST-TwolC Performance

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

In HFST, we have not yet implemented an *other-symbol* like the one in the Xerox TwolC presented by Karttunen [1992], the rule-transducers encode a number of unnecessary transitions. This slows down intersection, difference and determinization among other operations. It is probably the single most important factor slowing down HFST-TwolC. Like intersection, intersecting composition is also affected by the lack of an *other-symbol*, since intersecting composition is sensitive to the number of transitions in the rule-transducers.

7.3 Parallel Rules vs. Cascaded Rules

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

One possible reason for the speed-up is that HFST-LexC and HFST-TwolC are more constrained environments than the SFST utility `fst-compiler` by Schmid [2005], which is used for compiling the SFST version of the OMorFi analyzer. We suspect that the great liberty in constructing rules using SFST may tempt the user to indulge in unnecessarily unconstrained ways of expressing replacements with a very local area of application. This manifests itself among other things as an increased compile-time.

Converting the LexC and TwolC version of the Finnish lexicon from the SFST lexicon compiler files took approximately a week of manual work. While

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

7.4 The *Other-symbol*

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

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

7.5 Future Directions

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

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

8 Conclusions

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

Acknowledgments

...

References

[1983] Koskenniemi, K.: Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production. University of Helsinki, Department of General Linguistics (1983).

[1987] Karttunen, L., Koskenniemi, K., Kaplan, R.: A Compiler for Two-Level Phonological Rules. CSLI Publications (1987) `http://www2.parc.com/istl/members/karttune/publications/archive/twolcomp.pdf`.

[1992] Karttunen, L.: Two-Level Rule Compiler, Technical Report ISTL-92-2, Xerox Palo Alto Research Center (1992). `http://www.xrce.xerox.com/competencies/content-analysis/fssoft/docs/twolc-92/twolc92.html`.

[1993] Karttunen, L.: Finite-State Lexicon Compiler. Technical Report, ISTL-NLTT2993-04-02, Xerox Palo Alto Research Center (1993) Palo Alto, California.

[1994] Karttunen, L.: Constructing Lexical Transducers. The Proceedings of the 15th International Conference on Computational Linguistics COLING 94, I, 406–411 (1994)

[1997] Mohri, M.: Finite-state transducers in language and speech processing. Computational Linguistics 23(2), (1997)

[2002] Mohri, M., Riley. M.: An efficient algorithm for the n-best-strings problem. Proceedings of the International Conference on Spoken Language Processing 2002, ICSLP '02 (2002)

[2003] Beesley, K., Karttunen, L.: Finite State Morphology. CSLI Publications (2003). `http://www.fsmbook.com`.

[2004] Lombardy, S., Régis-Gianas, Y., Sakharovitch, J.: Introducing Vaucanson. Theoretical Computer Science 328, 77–96 (2004)

[2005] Schmid, H.: A programming language for finite state transducers. Proceedings of the 5th International Workshop on Finite State Methods in Natural Language Processing, FSMNLP 2005, (2005). Helsinki, Finland.

[2006] Yli-Jyrä, A., Koskenniemi, K.: Compiling Generalized Two-Level Rules and Grammars. Advances in Natural Language Processing, LNCS, 174–185 (2006)

[2007] Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., Mohri, M.: OpenFst: A general and efficient weighted finite-state transducer library. Proceedings of the Ninth International Conference on Implementation and Application of Automata, CIAA 2007, vol. 4783 LNCS, 11–23 (2007) `http://www.openfst.org`.

[2008] Pirinen, T.: Suomen kielen äärellistilainen automaattinen morfologia avoimen lähdekoodin menetelmin. Master's thesis, Helsingin yliopisto (2008)