UNIVERSITY OF HELSINKI

https://helda.helsinki.fi

HFST—Framework for Compiling and Applying Morphologies

Linden, Krister; Silfverberg, Miikka; Axelson, Erik; Hardwick, Sam; Pirinen, Tommi

Mahlow, Cerstin; Pietrowski, Michael

2011-08-26

http://hdl.handle.net/10138/29353

Linden, K, Silfverberg, M, Axelson, E, Hardwick, S & Pirinen, T 2011, HFST—Framework for Compiling and Applying Morphologies. in C Mahlow & M Pietrowski (eds), Systems and Frameworks for Computational Morphology. vol. Vol. 100, Communications in Computer and Information Science, vol. 100, Springer, pp. 67-85. https://doi.org/10.1007/978-3-642-23138-4_5

Downloaded from Helda, University of Helsinki institutional repository. https://helda.helsinki.fi
This is an electronic reprint of the original article.
This reprint may differ from the original in pagination and typographic detail.
Please cite the original version.

HFST—Framework for Compiling and Applying Morphologies

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

University of Helsinki
Department of Modern Languages
Unioninkatu 40 A
FI-00014 Helsingin yliopisto, Finland
{krister.linden, erik.axelson, sam.hardwick,
tommi.pirinen, miikka.silfverberg}@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 connects some of the most important finite-state tools for creating morphologies and spellers 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 key environments and operating systems. HFST also provides an opportunity to exchange transducers between different software providers in order to get the best out of each finite-state library.

Keywords: Finite-state libraries, finite-state morphology, natural language applications

1 Introduction

Including language technology in ordinary software applications can be both laborious and expensive if every language needs its own piece of software in addition to its lexicons and grammars. Standard interfaces to language modules have long been an effort equally promoted and hampered by large organizations by each having their own standard. However, with finite-state technology it is possible to go further and encode a range of language modules as finite-state transducers with a unified access interface.

To create the transducers, we still need lexicons and grammars for each language. Lexicons and grammars exist for close to a hundred languages with various degrees of coverage and elaboration. There are ongoing efforts to collect and list available sources in various software and data registries, e.g., VLO[1], META-SHARE[2] as well as more specific efforts for listing open-source finite-state descriptions on the HFST–Helsinki Finite-State Technology web site[3].

[1] www.clarin.eu/vlo/
[2] www.meta-net.eu/meta-share
[3] hfst.sf.net

Finite-state transducer (FST) technology has been well-known for several decades, and many packages of FST calculus exist. Some of them are available as open source. If they lose support, changing to another may easily mean redeveloping the language description.

The primary goals of HFST are to unify the field and create a framework for developing, compiling and applying morphologies, to create convergence and cooperation within the community which develops finite-state calculus and tools, to create a neutral platform where different implementations of the finite-state calculus can coexist and compete with each other, and to create a critical mass of research for improving the basic algorithms of the calculus, and compilation algorithms. HFST does this by providing an interface to an increasing number of software libraries for processing finite-state transducers in specialized ways. As far as possible, we have tried to avoid implementing yet another finite-state calculus. We have rather utilized existing free open source implementations, e.g., SFST by Helmut Schmid [22] and foma by Måns Huldén [10] for transducers without weights, as well as OpenFst by M. Riley, J. Schalkwyk, W. Skut, C. Allauzen and M. Mohri [2] for weighted transducers. A structural layout of how this compatibility is accomplished in HFST is available in section 2.

A second set of goals for HFST is to create and collect readily available open-source morphologies in order to provide a platform for basic high-performing natural language processing tools, to stimulate the production of free open-source software for compiling dictionaries, grammars and rules into FSTs, and to stimulate the production of language resources (e.g., dictionaries, grammars, rules) to be compiled into FSTs. HFST offers compatible open-source tools for compiling such language descriptions as well as storing them in formats that can be exchanged and further processed by the different finite-state libraries available in HFST, see section 3

An early adopter of the open-source paradigm was the SFST–Stuttgart Finite-State programming language (SFST-PL) which also has attracted developers of full-scale lexicons for various languages, e.g., German, Finnish, Turkish, or Italian. The benefits of HFST are demonstrated by using the foma library to implement the SFST-PL, which reduces the compile time to a fraction of the original. For further details on this, see section 4.

Over time, the Xerox commercial finite-state environment with compilers like TwolC, LexC and XFST has become popular and many academically developed language descriptions are available for these tools. This set of tools is now supported by HFST as a set of open source tools with an additional tool for composing parallel two-level rule sets with a lexicon. This is outlined in section 5.

We provide some insight into our compact and high-speed runtime transducers allowing processing speeds of more than 100,000 tokens per second using roughly one percent of the size of a corresponding uncompacted file, in section 6.

Some of the open-source application areas, where HFST is already in use, e.g., spell-checking through Voikko for OpenOffice, machine translation preprocessing via Apertium and part-of-speech tagging using parallel weighted transducers, are outlined in section 7.

HFST supports both commercial and open source applications. The tools and libraries of HFST are compatible with the GNU GPL license, which means that any FST

produced with HFST tools will remain under the licensing conditions of the input lexicons and rule sets. The HFST runtime format and runtime library are additionally released under the Apache license. This means that the HFST tools and the HFST runtime library can be used both for open source and proprietary projects. A further discussion of the impact of the HFST environment and some of the open-source morphologies currently available is provided in section 8.

2 Structural Layout

Finite-state transducer libraries such as SFST [22], foma [10] and OpenFst [2] all provide different ways for creating transducers, e.g., foma emulates the XFST formalism and SFST contains a compiler for its own regular expression formalism. The lack of common formalisms makes it difficult to compare the performance of the different libraries reliably, since they cannot be used for computing the same tasks.

One of the original goals in the HFST project was to provide a framework where it is possible to compare the performance of different finite-state transducer libraries on the same tasks. HFST 2.0 provided limited support for this by joining SFST and OpenFst under one interface. Regrettably, adding new libraries was cumbersome in HFST 2.0. To remedy this, HFST 3.0 was designed to make it practical to add both complete and partial implementations of transducer libraries and to use these for compiling LexC lexicons as well as TwolC, XFST and SFST-PL grammars.

Related to the goal of comparing performance, is the goal of combining algorithms from different transducer libraries in one task. This is now possible in HFST 3.0, where transducers can be converted between different underlying libraries. Thus it is possible to utilize well implemented algorithms from a variety of libraries in order to achieve faster compilation times. It is also possible to test the effect of a single transducer operation on the total compilation time of a task by switching between operations from different specialized libraries, i.e., it is possible to create specialized data-structures for certain operations in a separate library and use them for some special purpose operation without the need to re-implement a full set of well-researched basic transducer operations.

We first present the general structure of the HFST transducer class library and then we outline the procedure for adding a new or specialized library. We then mention the coding principles for exception handling in HFST and how new libraries can be linked and tested.

2.1 General Layout of HFST

Transducers in HFST are objects of the class `HfstTransducer`. The `HfstTransducer` class supports all the ordinary transducer operations like disjunction, composition and automaton determinization. `HfstTransducer` encapsulates the different transducer libraries under the HFST interface, i.e., the same code will work for all transducer libraries which are part of HFST. For example, the function `containment` in figure 1 works equally well for transducers whose underlying implementation is an SFST or a foma transducer.

```
HfstTransducer &containment
(HfstTransducer &center, const HfstTransducer &context)
{
  HfstTransducer context_copy(context);
  return center =
    context_copy.concatenate(center).concatenate(context).minimize();
}
```

Fig. 1. Function `containment` computes the language `context center context`, assigns it to center and returns a reference to center.

Internally `HfstTransducer` objects contain pointers to specific transducer library implementations. These are wrappers on the actual transducer libraries. Currently there are four wrappers for three libraries SFST, OpenFst and foma. The tropical weight semi-ring and log weight semi-ring in OpenFst have separate wrappers. An `HfstTransducer` can be initialized using any of the three transducer libraries, which are available, and it is possible to convert between the libraries at runtime.

A wrapper for a library consists of input and output stream classes for reading and storing binary transducers and a transducer class. The wrappers encapsulate implementation specific details in the different transducer libraries thus providing the HFST interface a unified way to manipulate objects from the different transducer libraries. For example, the wrappers for SFST are the classes `SfstInputStream`, `SfstOutputStream` and `SfstTransducer`.

Internally each `HfstTransducer` object `t` points to one of the implementations e.g., `SfstTransducer`. When `t.minimize()` is called, the `minimize` in `SfstTransducer` gets called.

In binary operations like `concatenate`, there is risk for a transducer type mismatch, since the `HfstTransducer` objects involved may have different types. In case they do, an exception is thrown. This can be caught and the transducers can be converted to a common type. The concatenation can then safely take place. However, in HFST 3.0, we have made a conscious choice not to convert transducers automatically in binary operations because this might lose information, e.g., when converting from weighted to unweighted formats.

2.2 Adding New Libraries to HFST

The task of adding a new transducer library `MyTransducerLibrary` to HFST can be broken into three subtasks.

1. Building the wrappers `MyTransducerLibraryInputStream`, `MyTransducerLibraryOutputStream` and `MyTransducerLibraryTransducer`.
2. Adding conversion functions to/from `MyTransducerLibraryTransducer` objects from/to the HFST internal format `HfstBasicTransducer`.
3. Adding necessary declarations in the master interface files `HfstTransducer.h` and `HfstTransducer.cc` in order to use the interface functions properly.

Unless `MyTransducerLibrary` has an alphabet implementation which associates string symbols with symbol numbers, such an alphabet also has to be created.

There is also support in the current HFST interface for transducers with weighted semi-rings which can not be represented as floating point numbers, although this may require implementing some new functions.

2.3 Coding Principles

Exceptional situations occur in computer programs when the user does something unexpected or there is a bug in the code. Examples of user-originating situations in a finite-state library include:

1. The user tries to read a binary transducer from a file that contains a text document or does not exist.
2. A transducer in the AT&T text format has a typo on one line and the line cannot be parsed.
3. The user calls a function without checking the preconditions, e.g., tries to extract all paths from a cyclic transducer.

Throwing an exception on such occasions gives the user a possibility to catch the exception and recover from the situation. In HFST version 2.0, exceptional situations were handled by printing a short message on the standard error stream and exiting with an error code. In HFST version 3.0, exceptions are classes that have a figurative name and contain an optional error message.

For the above scenarios, HFST will throw the following exceptions:

1. `NotTransducerStreamException` or `StreamCannotBeReadException` and, in the error message, the name of the file or stream.
2. `NotValidAttFormatException` and, in the error message, the line that could not be parsed.
3. `TransducerIsCyclicException`.

The user could react to the exceptions in the following ways:

1. Check that the file exists and contains transducers and try again with the correct file.
2. Fix the typo in the text format.
3. Call another function that limits the number of paths extracted from the transducer.

Exceptions are also used internally in the HFST library for reporting to a calling function that something unexpected happened. The calling function can handle the situation itself or inform the user and suggest what they should do, which means that the execution of the program may continue or terminate gracefully.

An `HfstFatalException` is thrown when it is unlikely that the user can handle the exception. The user should instead report the exception and its circumstances to the HFST developers, because the exception is essentially a bug that must be fixed. Some assertions are also used for internal checks. When an assertion fails the user should similarly report the failure as a bug.

2.4 Dynamic Linking to Underlying Libraries

There has been considerable progress in achieving the HFST goal of acting as a compatibility layer between different representations of finite-state transducers and, more importantly, the operations and formalisms (e.g., LexC, TwolC, XFST, SFST-PL) that have been implemented for them. HFST is now independent of any particular library and requires no custom extensions to the libraries it uses.

Previously, HFST relied on custom extensions to the libraries it supported, namely OpenFst and SFST, which made it necessary to statically link them into `libhfst`. This was obviously also rather restrictive in terms of new versions and different use cases, e.g., experimenting with local changes to the underlying libraries.

HFST 3.0 supports conditional compilation of all its elements that provide interfaces to underlying libraries, and dynamic linking is done to whichever libraries the user configures HFST 3.0 to use.

Due to the recent improvements, HFST 3.0 can also be built without any external libraries in which case the code supports only the operations for a simple internal representation and optimized-lookup, cf. section 6, i.e., building from text representation and fast lookup. If the user's own library is included, the code will also support compilation into the optimized-lookup format.

3 Data Formats

Each library is also expected to be able to handle its own external binary data format. For handling the external binary data of a specific library correctly with the HFST command line tools, the external binary data files are prepended with a header. Below, we outline the structure of this header.

Each of the underlying libraries of HFST is expected to have its own internal data format. As mentioned in the previous section, when adding a new library, conversion functions to and from the library-specific internal format need to be provided. We outline the procedure for this which normally is linear in time.

In addition, we may need to create transducers which deal with symbols that have not yet been specified in their alphabet. For this reason, we also need functions for harmonizing the alphabets between two or more transducers of the same format. We give the preconditions for this operation and refer the interested reader to the literature.

3.1 Transducer Binary Format

An HFST transducer in binary format consists of an HFST header followed by the back-end implementation in binary format[4]. In version 3.0, the header format is less error-prone than in the previous versions as it gives more information both for users of HFST, when seen on a screen or in a text editor in binary format, and for the HFST library itself.

The current header format is somewhat similar to foma where pieces of information are separated by newline characters to make them more readable. In HFST version 2.0,

[4] `https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstTransducerHeader`

we represented the properties of a transducer in a two-byte bit vector akin to the OpenFst header format. The type of the transducer and the existence of an optional alphabet in the transducer were also encoded with two characters in the beginning of the binary transducer which required familiarity with the specifications to interpret.

The new header format makes it easier to react to unexpected situations and inform the user, if necessary. When we read an HFST binary transducer, we first see whether an identifier 'HFST' is found. If not, we know that the user has given an incorrect file type and can throw an appropriate exception. Next we recognize the implementation type of the transducer. If the back-end transducer library is not linked to HFST, we can handle the situation by throwing another exception. Then we recognize the version of the header in order to process the rest of the header and the back-end implementation correctly. If the user has requested a verbose mode for a tool that is reading the transducer, it is also possible to print the name of each transducer before or after reading it.

3.2 Conversion Between Different Back-End Formats

In HFST version 3.0, the conversion between different back-end formats, i.e., SFST, foma or OpenFst with tropical and logarithmic semi-ring, is carried out through the prorpietary HFST transducer format, `HfstTransitionGraph`. The HFST internal format is a simple transition graph data type that consists of states (unsigned integers) and transitions between those states[5].

We have chosen to implement `HfstTransitionGraph` for two reasons. Firstly, it serves as an intermediate transducer format in conversions, thus reducing the number of conversion functions to $2 \times N$ from $N \times (N-1)$, where N is the number of different transducer back-end formats. Secondly, it is easy to implement functions for `HfstTransitionGraph` that allow the user to construct transducers from scratch and iterate through their states and transitions. Implementing such features for an existing transducer library, can sometimes require modifications of the library if the library is designed to be used on a higher level of abstraction, e.g., in SFST and foma, the functions that operate on states and transitions were protected and not well-documented.

`HfstTransitionGraph` is a class template with template parameters `T` and `W`. `T` defines the type of transition data that a transition uses and `W` the weight type that is used in transitions and final states. `HfstTransitionGraph` contains two maps. One maps each state to a set of the state's transitions that are of the type `class HfstTransition<class T>`. The other maps each final state to its final weight that is of type class `W`). Class `T` must use the weight type `W`. A state's transition `class HfstTransition<class T>` contains a target state and a transition data field that is of type class `T`.

Actually, `HfstTransitionGraph` is not a transducer but a more generalized transition graph that can contain many kinds of data in its transitions. Currently, the HFST library offers the specializations `HfstBasicTransducer` and `HfstBasicTransition` for `HfstTransitionGraph` and `HfstTransition`. These specializations are designed for weighted transducers. The weight class `W` is a float and the transition data class `T`

[5] `http://hfst.sourceforge.net/hfst3/index.html`

contains an input string, an output string and a weight of type float. The specializations `HfstBasicTransducer` and `HfstBasicTransition` are used when converting between different transducer back-end formats.

The class template `HfstTransitionGraph` is designed so that it can easily be extended to different kinds of transition data types. For example, if HFST tools are used in text-to-speech or speech-to-text conversion, the weights may be more complex and the symbol type of the transitions will probably be something else than strings.

3.3 Alphabet

The alphabet of a transducer consists of all symbols (strings) that are known to that transducer. The alphabet includes all symbols that occur or have occurred in the transitions of the transducer unless explicitly removed from the alphabet. If we apply a binary operation (e.g., disjunction or composition) on transducers A and B, the resulting transducer's alphabet will include all symbols that were in the alphabets of A and B.

In HFST version 2.0, alphabets were designed to be external to the transducer, and the interface did not offer any convenient way for the user to access a transducer's internal representation of the alphabet. It was up to the back-end implementation to take care of the alphabet of an individual transducer. In SFST the transducers always have an explicit alphabet, but in OpenFst their use is optional.

In HFST version 3.0, we need to be aware of transducer-specific alphabets because two new special symbols are included, `unknown` and `identity`. These special symbols are a part of the Xerox Finite-State Tool (XFST) formalism [5] and they are also implemented in foma [10]. The `unknown` and `identity` symbols are useful when we want to refer to all symbols that are not currently known to a transducer but which the transducer can later become aware of.

Supporting `unknown` and `identity` symbols in all HFST back-end implementations has enabled us to provide an XFST compiler that can be used with all back-end implementations. In this way, we can offer users of HFST a new formalism for regular expressions in addition to the one available in SFST. As the extension is already implemented in foma, we only needed to consider SFST and OpenFst in HFST 3.0.

Aside from keeping track of all symbols known to an individual transducer, we also have to expand each transition involving `unknown` and `identity` symbols into a set of transitions every time we apply a binary operation on two transducers. This is because the transducer becomes aware of new symbols that are no longer unknown and thus no longer included in the `unknown` or `identity` symbols. Fortunately, this expansion can be done before the operation itself (and for composition before and after the operation itself), i.e., it is not necessary to make changes in the operations of the back-end transducer libraries. The library operations can and will handle the special symbols just like any ordinary symbols.

First we iterate through the alphabets of both transducers and find out which symbols in the alphabet of one transducer are not found in the alphabet of the other transducer and vice versa. Then we add beside each transition involving the `unknown` or `identity` symbols a set of transitions, where `unknown` and `identity` symbols are re-

placed with all symbols that the transducer just became aware of. For more information on how to expand special symbols, see [10] and [5].

It is also possible to switch off the handling of special symbols if we know for sure that they are not used in the transducers. In this way, we can optimize performance for instance for the tool `hfst-sfstpl2fst` that processes the SFST-PL formalism which does not support the unknown or identity symbols.

4 SFST Programming Language Compatibility

The performance of HFST has improved from version 2.0 to 3.0. We compiled two finite-state morphologies in the SFST-PL format with HFST versions 2.0 and 3.0. The morphologies were OMorFi [17] for Finnish and Morphisto [27] for German. Table 1 shows the compilation times for both morphologies with different back-end implementations with both versions of HFST. Note that the foma implementation was not available in version 2.0.

Table 1. Compilation times for Finnish and German morphologies with different HFST versions. The times are expressed in minutes and seconds.

Back-End	HFST	Finnish	German
SFST	2.0	25:16	107:47
	3.0	5:02	6:39
OpenFst	2.0	7:54	6:23
	3.0	6:51	6:28
foma	2.0	—	—
	3.0	1:49	1:29

We can clearly see that the compilation time has improved dramatically for the SFST implementation. This is mainly because the new version of SFST, 1.4.2, uses Hopcroft's minimization algorithm [9] instead of Brzozowski's [6]. We noticed that the Brozowski minimization algorithm hampered the performance already when we were testing HFST version 2.0; OpenFst was clearly faster because it used the Hopcroft algorithm. Based on this observation, Helmut Schmid improved SFST by writing a minimization function using the Hopcroft algorithm.

When comparing the compilation times for OpenFst, we see that the Finnish morphology compiles faster but the German one slightly slower on HFST version 3.0 than on version 2.0. This is because there are two main factors that contribute to the difference in performance. Firstly, we are currently using OpenFst version 1.2.7 that is faster than the previous versions. Secondly, in HFST version 3.0 we no longer use a global number-to-symbol encoding for all transducers during one session. Every time we perform a binary operation on two transducers, we harmonize the encodings of the transducers. Nevertheless, it seems that the newer, more efficient version of OpenFst mostly compensates for this additional effort caused by harmonization.

We did not have the foma implementation of the SFST-PL available in HFST version 2.0, but it is evident that it is much faster than the other implementations in either version of HFST. Foma does not use a global symbol-to-number encoding in its transducers either, but it still performs well. This is evidence that symbol harmonization is not a big factor in the compilation times of morphologies.

5 Xerox Compatibility

Among the goals of the HFST framework has always been to retain legacy support for the Xerox line of tools for building finite-state morphologies [5]. Optimally, the tools should be familiar to end-users converting from the Xerox tools. For this purpose we have aimed to create clones of the most important Xerox tools as accurately as possible. Previous open-source implementations of Xerox tool clones have included LexC and TwolC [13] and LexC and XFST [10]; in HFST 3.0 we have combined these contributions into one uniform package capable of handling the full line of Xerox tools for morphology. For the most part, end-users will require no other familiarization than changing program names in order to start using HFST tools for their Xerox-style language description needs.

The implementation of the XFST scripting language makes heavy use of the new foma back-end, which already had a good coverage of XFST features. The only additions in HFST are the ones required for interoperability between other back-ends and HFST internals. Particular care was taken not to duplicate the work already present in foma and its tools. Similarly HFST's LexC parsing engine was replaced by the faster LexC parser in foma, again with HFST interoperability tweaks straddling the gaps.

For practical examples of specific previously implemented Xerox style finite-state language descriptions we provide a wiki-based web page[6]. Another repository of such language descriptions is located at the University of Tromsø's subversion repository[7]. Of these, the morphologies for the Sámi languages and Greenlandic are regularly used in regression and stress tests of the HFST tools.

For specific functionalities, the Xerox tools perform various kinds of special processing of finite-state transducers beyond the range of standard finite-state algorithms. A prominent example of this is the handling of special symbols such as flag diacritics [4], which would require support from the underlying libraries for many finite-state operations to work as they do in Xerox tools when using `flag-is-epsilon` and `obey-flags` settings. HFST tools provide support for such options, and provide fall-back processing where back-end libraries lack support for the required operations. Fall-back support commonly involves converting the back-end library internal transducers to the HFST internal format, calculating the operation and converting the transducer back to the back-end library format.

[6] `https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstExamples`
[7] `http://divvun.no/doc/infra/anonymous-svn.html`

5.1 Intersecting Composition

Intersecting composition is used for applying a grammar of two-level rules to a two-level lexicon. The result of the operation is, e.g., a morphological analyzer mapping word forms to analyses. Compiling the analyzer using conventional methods requires computing the intersection of the rule transducers. This may lead to a prohibitively large intermediate result. Intersecting composition avoids computing the entire intersection of the rules thus reducing both memory and time requirement. The operation was introduced by Karttunen [11] and later extended to weighted transducers in HFST 2.0 by Silfverberg and Lindén [23].

The intersecting composition operation was implemented in HFST 2.0, but we used techniques adopted from OpenFst [2] to improve the implementation, and the current implementation is significantly faster than the old one. The current implementation computes a lazy pairwise intersection of the rule transducers. The lexicon can be composed with this intersection using a standard composition algorithm.

Previous Implementation. The implementation of intersecting composition in HFST 2.0 can be characterized as the composition of the lexicon transducer L with a structure P containing all rule transducers. For simplicity, we assume that the lexicon and rule transducers are deterministic.

Outwards the structure P resembles an ordinary transducer with states and transitions. The states of P internally correspond to vectors of rule transducer states, which we call state configurations. The vectors have as many indexes as there are rules and each rule corresponds to a unique index, where its state is stored. For example, the start state of P corresponds to the vector containing the start states of the rules.

Initially only the start state of P is computed. More states in P are computed according to the transitions in L. For example, the lexicon L might have a transition with output-symbol `a` in its initial state. In order to compute the intersecting composition, it would be necessary to create transitions and corresponding target states in P for all symbol pairs `a:b`, where each of the rules has transitions from its initial state with symbol pair `a:b`. Outwards P would have one target state t for the transition with pair `a:b` from its initial state. Internally t would correspond to a configuration of the target states of the transitions with symbol pair `a:b` in each of the individual rules.

There is no caching of transitions in the states of P. Thus the transitions in states have to be recomputed every time the algorithm visits a given configuration of rule states. This requires more work than if the transitions were cached, since there usually exist state configuration which are very frequently visited.

Even if the transitions in states were cached, this implementation would still be suboptimal, since a new state configuration always requires re-examining the transitions in all rules. This is true even if only one rule state differs from a previous configuration.

Current Implementation. We note that phonological two-level rules usually track sound changes in fairly specific contexts. This means that when composing two-level rule transducers with a lexicon, the rules will occupy a limited state set during the majority of the time of the composition. The current implementation of intersecting composition capitalizes on this property.

Instead of a parallel lazy intersection like we used in HFST 2.0, we recursively build an intersection of the rules by intersecting them lazily pairwise. The first and second rule are intersected, the result is intersected with the third rule and so on. The HFST 3.0 implementation caches the transitions of a given state pair, so there is no need to recompute them when the state is revisited.

This leads to significant improvement in performance, see table 2. The improvement results from caching transitions, but it is improved by the fact that computing the transitions in a previously unseen state configuration does not require recomputing the transitions in all of the rules. For example, if rule number n moves to a new state with symbol pair `a:b`, but the rest of the rules remain in a familiar state configuration, we only need to recompute the transitions of the rules having a greater index than n. This derives from the fact that we have already cached the target state of the subset of rules 1 to $n-1$ in their lazy intersection structure.

Like the old implementation, the current implementation of intersecting composition is equivalent with Xerox style flag diacritics and `identity` symbols.

Table 2. Runtimes for intersecting composition of the Finnish and Northern Sámi morphological analyzers in HFST 2.0 and HFST 3.0.

Language	HFST 2.0	HFST 3.0
North Sámi	364.2 s	63.4 s
Finnish	4.1 s	2.6 s

6 Runtime and Optimized Lookup

Optimized-lookup is an HFST-specific binary format for finite-state transducers providing fast lookup. First documented in [24], its implementation has evolved somewhat to meet the requirements of specific applications. The format has also found new application in on-the-fly operations in transducers, e.g., composing lookup for spell-checking and correction, see section 7.2.

6.1 Implementation and Integration in HFST 2.0 and HFST 3.0

Optimized-lookup was supported in HFST 2.0 by the standalone utilities `hfst-lookup-optimize` (compilation) and `hfst-optimized-lookup` (non-tokenizing lookup). This was partly in service of the goal of giving optimized-lookup the widest possible range of uses; `hfst-optimized-lookup` was released under the Apache License [3], whereas HFST 2.0 proper was released under the potentially more restrictive GNU Lesser Public License [7].

Provision was later made for both unweighted and weighted (with log weights) transducers, and flag diacritics [5], see also section 6.3.

Other Implementations and Applications. Demonstrations of the lookup facility were also produced in Java and Python, two popular and accessible programming languages, in the hope of facilitating and spreading use of the format. This effort bore fruit in the incorporation of the Java code in a project for anonymizing identities in legal documents at the Aalto University in Helsinki.

The format saw additional uses and implementations over the course of furthering research goals and maintaining HFST 2.0. Hyphenators and spell checking transducers were primarily used in this format for its speed, and in 2010 a Google Summer of Code[8] project by Brian Croom produced `hfst-proc`, a tokenizing lookup application for optimized-lookup which was put to use in various text stream processing scripts (e.g., `-analyze` for analysis, `-generate` for generation and `-hyphenate` for hyphenation).

HFST 3.0. Originally compilation to the format was only possible from the SFST and OpenFst formats, and as uses and applications proliferated, it became desirable to provide some API access to optimized-lookup in the HFST library. In HFST 3.0 this has been accomplished by implementing compilation from the HFST internal transducer format, allowing for a great degree of integration with HFST 3.0 supported tools.

6.2 Index Table Compaction

The crucial idea behind optimized-lookup is Liang compaction, as described in [24]. It allows for the representation of a transducer as a lookup table, with entries for each symbol in the alphabet for each state in the transducer, without growing to the prohibitive sizes such a design would imply, i.e., a multiple of the number of states and the number of symbols for the state indexing table alone. Liang in his PhD thesis on hyphenation [12] did not specify a generalized compaction scheme, only the requirements for its correctness. In realistic transducers, finding the optimal compaction is in any case computationally infeasible, and consequently some approaches to producing a "good enough" compaction have been attempted.

For the purposes of this article, the task of compacting the index table may be summarized as the following. Given N arrays (one for each state) s of length L (the size of the alphabet), the entries of which are 0 (representing an unused entry) or 1 (representing a used entry), calculate a list of starting indexes $I_1, I_2 \ldots I_N$ such that a result array with entries

$$R_i = \sum_{p,q} s_p(q)[I_p + q = i] \tag{1}$$

will also have entries 0 or 1. The brackets [and] are Iverson brackets; the value of the bracketed expression is 1 if the condition is true and 0 if it is false. This array will contain all N arrays superimposed on each other in such a way that each entry in the result array is used by no more than one of the original arrays. An optimally compacted index table corresponds to the shortest result array R.

A simple strategy is to iterate through the arrays in some order, assigning the lowest possible starting index to each one. For transducers of an appreciable size this process

[8] `http://code.google.com/soc/`

can become slow, as the result array will typically have some zeros in practically all its regions, so a large number of possibilities have to be checked. This problem can be mitigated by applying a head filter, disregarding the largest region of the result array $R(1 \ldots r)$ with that region having a density of 1 entries greater than some predefined limit. A limit of 1.0 corresponds to having no filter at all; in practice limits in the region $0.8 \ldots 0.9$ have proved reasonable, see table 3.

Table 3. Index table sizes for various values of the head filter limit using in-order traversal as applied to the Morphalou project's French morphology and released on the HFST site on 2010-04-14.

Head filter limit	Number of index entries
(No compaction)	4,541,879
0.0	1,107,321
0.1	408,843
0.3	206,152
0.5	170,789
0.7	155,703
0.8	148,740
0.9	140,795
1.0	135,285

For the order in which the arrays are traversed, ordering the states from greatest density of 1 entries to lowest and simply ordering them in numerical order have been tried. These approaches don't appear to produce dramatically different results[9]; also in theory, either approach could produce better results than the other.

Work is ongoing in finding the best practical filter strategies, filter limits and traversal orders, and potentially different compaction strategies.

A representation of the index table with no compaction would require $N \times L$ entries. For the sake of comparison, in the case of an analyzing transducer from OmorFi[10], a Finnish finite-state morphology, this is $203,851 \times 155 = 31,596,905$ and would produce a binary of almost 200 megabytes, whereas the compacted index table has 364,980 entries, or about one percent of the uncompacted form, for a binary of 7 megabytes.

6.3 Flag Diacritics and Related Optimization Tricks

Support for flag diacritics was added to the `optimized-lookup` utility with an eye to efficiency, prompting some refinements to the HFST implementation of the format itself. Flag diacritics are parsed prior to lookup and during it they restrict the lookup search tree. This is critical for speed, as the alternative of calculating all the outputs first and then removing outputs with conflicting flag diacritics can, in the case of transducers with liberal use of flags, involve several times the work.

[9] Of course, in practice these orderings will often be similar.

[10] We used the binary released on the HFST site on 2010-10-14.

For the purposes of traversing transitions, flag diacritics are essentially a special case of the epsilon symbol. If the configuration of flags that have been traversed up to a certain point allow it, each transition with a flag diacritic is traversed without reading an input symbol. With this in mind, it was desirable to avoid checking for transitions with each flag symbol in each state. The optimized-lookup format was therefore amended to treat flag diacritics as epsilon for purposes of constructing the index table and to list their transitions out of the normal order, after the epsilon transitions. Thus it is possible to check only those flag transitions that are present in a given state.

This allows further reductions in the size of the index table; as the indexes for flag diacritics are no longer in use, it is possible to reduce the effective size of the input alphabet by the number of different flag diacritic operations. These improvements may appear minor, but are not insignificant in transducers that make substantial use of flag diacritics. In the previously discussed version of OmorFi, we achieved a reduction of 12.4% in index table size.

7 Application Areas

After the initial release of the HFST platform, it has been used in several end-product applications. The two most prominent uses are as a part of the rule-based machine translation platform Apertium[11] and the spell-checking library Voikko[12]. Both of these linguistic applications benefit hugely from the fact that there were previous language descriptions available written in the Xerox finite-state morphology formalism, and integrating HFST in these applications gave developers of those language descriptions a direct conversion path to two application types that had not been available in the Xerox framework. Since both applications, as well as the HFST framework, are free/libre open-source software, the integration of the existing language descriptions in the projects was possible.

One of the most pressing reasons for extending finite-state support to these applications is the lack of language support and the lack of a theoretically well-motivated open-source option for language support for morphologically more complex languages in the above-mentioned applications. For example in the field of spell-checking the theoretical upper-bound for hunspell—de facto standard in the open-source market—is a mere 4 affixes. For polysynthetic languages like Greenlandic, it simply is not possible to precompute enough affix and stem combinations, as has been done with e.g., Hungarian. The Xerox style finite-state morphology demonstrably supports at least Hungarian and a wide variety of other morphologically varied languages [5].

Another rationale for extending finite-state methods to the application areas is that the efficiency and expressiveness of finite-state automata is well-known and has been researched e.g., in [1], which makes it a good choice for various text-processing tasks.

[11] `http://apertium.sf.net`
[12] `http://voikko.sf.net`

7.1 Apertium Interoperability—Corpus Processing Tools and I/O Formats

In Apertium, the finite-state automata are used to perform morphological analysis and generation for both parsing running text and generating the translations after performing a mid-shallow rule-based transfer. The HFST software is only one possible morphological analyzer, so the crucial part for inclusion was to get the HFST analyzer to work as the competition. This included two functions: reliable tokenization based on the dictionary data and support for Apertium I/O formats. The contribution of the corpus processing functionality is contained in a tool called `hfst-proc` also included in the HFST toolkit. The name is influenced by similar corpus processing tools in other toolkits, specifically `cg-proc` from VISLCG3[13] and `lt-proc` from Apertium itself.

For tokenization the FST-based dictionaries are useful, since the process of analysis and lookup can both be performed by basic FST traversal. The specific implementation of analysis and tokenization with a single FST traversal was implemented as a Google Summer of Code project, based on previous studies on the topic [8]. The basic programming logic of the automata traversal for the longest match is trivially extended by the processing of flag diacritics and weights.

The I/O format requirements for the Apertium platform are based on the needs to translate existing documents containing all kinds of markup and rich text formats, such as HTML for web pages or MediaWiki codes from Wikipedia. To achieve this, Apertium uses text encoding and decoding mechanisms and an interchange format called Apertium stream format, whose input and output was implemented in the HFST corpus processing tools.

7.2 Voikko and HFST Based Spell-Checker Formulation

The application of finite-state morphologies in spell-checking applications is also based on a new development of finite-state algorithms. The application framework including HFST spell-checkers is the Voikko library, which provides spell-checkers for OpenOffice.org/LibreOffice, the GNOME desktop (via enchant), Mac OS X (via SpellService) and the Mozilla application suite.

The finite-state formulation of a spell-checking system was developed based on previous research. In this research it has been shown that a finite-state based natural language description is usable as spell-checker with specialized fuzzy traversal algorithms [16,10] or by using a special (weighted) two-tape automaton as an error model and regular finite-state composition to map misspelled words to their possible corrections [21,20].

The basic finding here is that typical finite-state language descriptions are usable as spell-checking dictionaries with minor to no modifications. Furthermore it has been shown that existing non-finite-state spell-checking dictionaries from hunspell and myspell can be converted into finite-state form [19] providing full backwards compatibility for traditional spell-checking systems.

Furthermore, we have optimized the application of error models when suggesting corrections by applying a three-way composition with both the dictionary, the error

[13] `http://beta.visl.sdu.dk/cg3.html`

model and the misspelled word in one operation. This significantly reduces the space and time requirements by leaving many of the impossible intermediate results uncalculated.

7.3 Statistical Part-of-speech Tagging using Weighted Finite-State Transducers

HFST 3.0 has also been applied to part-of-speech tagging of the Finnish, Swedish and English Europarl corpora [25] as well as the Wall Street Journal corpus [26]. Silfverberg and Lindén implemented first order Hidden Markov Models (HMM) as sets of parallel weighted finite-state transducers using HFST 3.0 tools. Like standard HMMs, their tagger used tag sequences. In addition they included lemmas in their models. Part-of-speech tagging was accomplished by intersecting composition of a sentence automaton with the set of tagger transducers.

8 Discussion

In parallel with HFST, there is the OMor project[14] for creating and/or compiling open-source morphological analyzers for Finnish, Swedish, French, German and English. OMorFi [18] is a large-scale Finnish open-source morphological transducer lexicon based on words from a dictionary with inflectional codes as well as patterns for compounding and derivation. It is available both for the SFST-PL as well as a Xerox-style two-level morphology. The Divvun project in Norway has created two-level morphological analyzers for Northern and Lule Sámi using Xerox tools. In addition, there are several projects having developed SFST-PL lexicons for German[15], Italian[16], Turkish[17], etc. Also close to a hundred HFST spellers [19] with an improved error correction mechanism already exist having been compiled from Hunspell sources and large corpora. The effort to collect existing morphological descriptions for various languages is on-going.

A concrete outcome of the current HFST environment is that it is now possible to take a lexicon developed with e.g., XFST or Hunspell and weight it with material from a specialized corpus in order to create a tailored speller for a given domain. This can all be accomplished in the course of an afternoon after which we can start using it in e.g., OpenOffice.

As future work, we are investigating how to extend the morphological analyzers into finite-state implementations of constraint grammars[18] and other dependency-related tagger and grammar formalisms using both statistical and rule-based approaches with weighted finite-state transducers.

9 Conclusion

In this article, we have described the structural layout of HFST–the Helsinki Finite-State Technology library, how it connects some existing finite-state libraries and how it can

[14] `http://www.ling.helsinki.fi/kieliteknologia/tutkimus/omor/`
[15] Morphisto [27]
[16] `http://dev.sslmit.unibo.it/linguistics/morph-it.php`
[17] TRmorph–`http://www.let.rug.nl/~coltekin/trmorph/`
[18] `http://beta.visl.sdu.dk/cg3.html`

accommodate additional libraries for finite-state algorithms and applications. Facilitating data exchange between finite-state implementations is important for processing and enhancing language descriptions created with different tools and formalisms. In this article, we focused on two finite-state programming language environments, i.e., the SFST Programming Language and the Xerox tools for creating morphological descriptions, and showed that the HFST toolkit can cover a wider range of morphologies and applications than the original finite-state environments. We have also demonstrated that the cross-usage of finite-state programming language front-ends and finite-state library back-ends can provide significant reductions in processing time.

Acknowledgments We would like to thank the contributors of the open source community for their support and excellent test applications. Special thanks go to Sjur Moshagen of Divvun, Francis Tyers of Apertium, and Harri Pitkänen of Voikko. We are also grateful to FINCLARIN for making HFST possible.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools with Gradiance. Addison-Wesley Publishing Company, USA, 2nd edn. (2007)
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 Ninth International Conference on Implementation and Application of Automata, (CIAA 2007). LNCS, vol. 4783, pp. 11–23. Springer, Heidelberg (2007), `http://www.openfst.org`
3. Apache Software Foundation: Apache License, Version 2.0, `http://www.apache.org/licenses/LICENSE-2.0.html`
4. Beesley, K.R.: Constraining separated morphotactic dependencies in finite-state grammars. In: Karttunen, L., Oflazer, K. (eds.) Proceedings of the International Workshop on Finite State Methods in Natural Language Processing. pp. 118–127. Association for Computational Linguistics, Morristown, NJ, USA (1998)
5. Beesley, K.R., Karttunen, L.: Finite State Morphology. CSLI publications (2003)
6. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM 11, 481–494 (October 1964)
7. Free Software Foundation: GNU Lesser General Public License, Version 3, `http://www.gnu.org/licenses/lgpl.html`
8. Garrido-Alenda, A., Forcada, M.L., Carrasco, R.C.: Incremental construction and maintenance of morphological analysers based on augmented letter transducers (2002)
9. Hopcroft, J.E.: An n log n algorithm for minimizing states in a finite automaton. Tech. rep., Stanford University, Stanford, CA, USA (1971)
10. Huldén, M.: Fast approximate string matching with finite automata. Procesamiento del Lenguaje Natural 43, 57–64 (2009)
11. Karttunen, L.: Constructing lexical transducers. In: The Proceedings of the 15th International Conference on Computational Linguistics COLING94. pp. 406–411. ACL, Morristown, New Jersey, USA (1994)
12. Liang, F.M.: Word hy-phen-a-tion by com-pu-ter. Ph.D. thesis, Stanford University (1983), `http://www.tug.org/docs/liang/`
13. Lindén, K., Silfverberg, M., Pirinen, T.: HFST tools for morphology—an efficient open-source package for construction of morphological analyzers. In: Mahlow and Piotrowski [14], pp. 28–47

14. Mahlow, C., Piotrowski, M. (eds.): Workshop on Systems and Frameworks for Computational Morphology, SFCM 2009, Zürich, Switzerland, September 2009, Proceedings, CCIS, vol. 41. Springer, Heidelberg (2009)
15. NODALIDA 2011, the 18th Nordic Conference of Computational Linguistics, Riga, May 11–13, 2011, Proceedings (2011)
16. Oflazer, K.: Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. Computational Linguistics 22(1), 73–89 (1996)
17. Pirinen, T.: Suomen kielen äärellistilainen automaattinen morfologinen analyysi avoimen lähdekoodin menetelmin. Master's thesis, Helsingin yliopisto (2008), http://www.helsinki.fi/~tapirine/gradu/
18. Pirinen, T.: Modularisation of Finnish finite-state language description–towards wide collaboration in open source development of a morphological analyser. In: Nodalida [15]
19. Pirinen, T.A., Lindén, K.: Building and using existing hunspell dictionaries and TEX hyphenators as finite-state automata. In: Proccedings of Computational Linguistics – Applications, 2010. pp. 25–32. Wisła, Poland (2010), http://www.helsinki.fi/~tapirine/publications/Pirinen-cla-2010.pdf
20. 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)
21. 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, London, UK (2002)
22. Schmid, H.: A programming language for finite state transducers. In: Proceedings of the 5th International Workshop on Finite State Methods in Natural Language Processing (FSMNLP 2005). pp. 308–309. Helsinki (2005)
23. 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)
24. 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
25. Silfverberg, M., Lindén, K.: Part-of-speech tagging using parallel weighted finite-state transducers. In: Loftsson, H., Rögnvaldsson, E., Helgadóttir, S. (eds.) Advances in Natural Language Processing – Proceedigns of the 7th International Conference on NLP, IceTAL 2010. vol. 6233, pp. 369–381. Reykjavik, Iceland (2010)
26. Silfverberg, M., Lindén, K.: Combining statistical models for POS tagging using finite-state calculus. In: Nodalida [15]
27. Zielinski, A., Simon, C.: Morphisto: Service-oriented open source morphology for German. In: Mahlow and Piotrowski [14], pp. 64–75