

Building and Using Existing Hunspell Dictionaries and T_EX Hyphenators as Finite-State Automata

(Authors' pre-print draft version)

Tommi A Pirinen, Krister Lindén

University of Helsinki,

Department of Modern Languages

Unionkatu 40, FI-00014 University of Helsinki, Finland

Email: {tommi.pirinen,krister.linden}@helsinki.fi

Abstract—There are numerous formats for writing spell-checkers for open-source systems and there are many descriptions for languages written in these formats. Similarly, for word hyphenation by computer there are T_EX rules for many languages. In this paper we demonstrate a method for converting these spell-checking lexicons and hyphenation rule sets into finite-state automata, and present a new finite-state based system for writer's tools used in current open-source software such as Firefox, OpenOffice.org and enchant via the spell-checking library voikko.

I. INTRODUCTION

CURRENTLY there is a wide range of different free open-source solutions for spell-checking and hyphenation by computer. For hyphenation the ubiquitous solution is the original T_EX algorithm described in [1]. The most popular of the spelling dictionaries are the various instances of *spell software, i.e. ispell¹, aspell², myspell and hunspell³ and other *spell derivatives. The T_EX hyphenation patterns are readily available on the Internet to cover some 49 languages. The hunspell dictionaries provided with the OpenOffice.org suite cover 98 languages.

The program-based spell-checking methods have their limitations because they are based on specific program code that is extensible only by coding new features into the system and getting all users to upgrade. E.g. hunspell has limitations on what affix morphemes you can attach to word roots with the consequence that not all languages with rich inflectional morphologies can be conveniently implemented in hunspell. This has already resulted in multiple new pieces of software for a few languages with implementations to work around the limitations, e.g. emberek (Turkish), hspell (Hebrew), uspell (Yiddish) and voikko (Finnish). What we propose is to use a generic framework of finite-state automata for these tasks. With finite-state automata it is possible to implement the spell-checking functionality as a one-tape weighted automaton containing the language model and a two-tape weighted automaton containing the error model. This also allows simple use of unigram training for optimizing spelling suggestion

results [2]. With this model, extensions to context-based n-gram models for real-word spelling error problems [3] are also possible.

We also provide a method for integrating the finite-state spell-checking and hyphenation into applications using an open-source spell-checking library voikko⁴, which provides a connection to typical open-source software, such as Mozilla Firefox, OpenOffice.org and the Gnome desktop via enchant.

II. DEFINITIONS

In this article we use weighted two-tape finite-state automata—or weighted finite-state transducers—for all processing. We use the following symbol conventions to denote the parts of a weighted finite-state automaton: a transducer $T = (\Sigma, \Gamma, Q, q_0, Q_f, \delta, \rho)$ with a semi-ring $(S, \oplus, \otimes, \bar{0}, \bar{1})$ for weights. Here Σ is a set with the input tape alphabet, Γ is a set with the output tape alphabet, Q a finite set of states in the transducer, $q_0 \in Q$ is an initial state of the transducer, $Q_f \subset Q$ is a set of final states, $\delta : Q \times \Sigma \times \Gamma \times S \rightarrow Q$ is a transition relation, $\rho : Q_f \rightarrow S$ is a final weight function. A successful path is a list of transitions from an initial state to a final state with a weight different from $\bar{0}$ collected from the transition function and the final state function in the semi-ring S by the operation \otimes . We typically denote a successful path as a concatenation of input symbols, a colon and a concatenation of output symbols. The weight of the successful path is indicated as a subscript in angle brackets, $input:output_{<w>}$. A path transducer is denoted by subscripting a transducer with the path. If the input and output symbols are the same, the colon and the output part can be omitted.

The finite-state formulation we use in this article is based on Xerox formalisms for finite-state methods in natural language processing [4], in practice lexc is a formalism for writing right linear grammars using morpheme sets called lexicons. Each morpheme in a lexc grammar can define their right follower lexicon, creating a finite-state network called a *lexical transducer*. In formulae, we denote a lexc style lexicon named X as Lex_X and use the shorthand notation $Lex_X \cup input:output Y$ to denote the addition of a lexc string or morpheme, $input:output Y$; to the LEXICON X . In

¹<http://www.lasr.cs.ucla.edu/geoff/ispell.html>

²<http://aspell.net>

³<http://hunspell.sf.net>

⁴<http://voikko.sf.net>

the same framework, the twolc formalism is used to describe context restrictions for symbols and their realizations in the form of parallel rules as defined in the appendix of [4]. We use $Twol_Z$ to denote the rule set Z and use the shorthand notation $Twol_Z \cap a:b \leftrightarrow l e f t_r i g h t$ to denote the addition of a rule string $a:b \leftrightarrow l e f t_r i g h t$; to the rule set Z , effectively saying that $a:b$ only applies in the specified context.

A spell-checking dictionary is essentially a single-tape finite-state automaton or a language model T_L , where the alphabet $\Sigma_L = \Gamma_L$ are characters of a natural language. The successful paths define the correctly spelled word-forms of the language [2]. If the spell-checking automaton is weighted, the weights may provide additional information on a word's correctness, e.g. the likelihood of the word being correctly spelled or the probability of the word in some reference corpus. The spell-checking of a word s is performed by creating a path automaton T_s and composing it with the language model, $T_s \circ T_L$. A result with the successful path $s_{<W>}$, where W is greater than some threshold value, means that the word is correctly spelled. As the result is not needed for further processing as an automaton and as the language model automaton is free of epsilon cycles, the spell-checking can be optimized by performing a simple traversal (lookup) instead, which gives a significant speed-advantage over full composition [5].

A spelling correction model or an error model T_E is a two-tape automaton mapping the input text strings of the text to be spell-checked into strings that may be in the language model. The input alphabet Σ_E is the alphabet of the text to be spell-checked and the output alphabet is $\Gamma_E = \Sigma_L$. For practical applications, the input alphabet needs to be extended by a special any symbol with the semantics of a character not belonging to the alphabet of the language model in order to account for input text containing typos outside the target natural language alphabet. The error model can be composed with the language model, $T_L \circ T_E$, to obtain an error model that only produces strings of the target language. For space efficiency, the composition may be carried out during runtime using the input string to limit the search space. The weights of an error model may be used as an estimate for the likelihood of the combination of errors. The error model is applied as a filter between the path automaton T_s compiled from the erroneous string, $s \notin T_L$, and the language model, T_L , using two compositions, $T_s \circ T_E \circ T_L$. The resulting transducer consists of a potentially infinite set of paths relating an incorrect string with correct strings from L . The paths, $s : s_{<w_i>}$, are weighted by the error model and language model using the semi-ring multiplication operation, \otimes . If the error model and the language model generate an infinite number of suggestions, the best suggestions may be efficiently enumerated with some variant of the n-best-paths algorithm [6]. For automatic spelling corrections, the best path may be used. If either the error model or the language model is known to generate only a finite set of results, the suggestion generation algorithm may be further optimized.

A hyphenation model T_H is a two-tape automaton mapping input text strings of the text to be hyphenated to possibly hyphenated strings of the text, where the input alphabet, Σ_E , is the alphabet of the text to be hyphenated and the output alphabet, Γ_E , is $\Sigma_E \cup H$, where H is the set of symbols marking hyphenation points. For simple applications, this equals hyphens or discretionary (soft) hyphens $H = -$. For more fine-grained control over hyphenation, it is possible to use several different hyphens or weighted hyphens. Hyphenation of the word s is performed with the path automaton T_s by composing, $T_s \circ T_H$, which results in an acyclic path automaton containing a set of strings mapped to the hyphenated strings with weights $s : s_{<w_h>}$. Several alternative hyphenations may be correct according to the hyphenation rules. A conservative hyphenation algorithm should only suggest the hyphenation points agreed on by all the alternatives.

III. MATERIAL

In this article we present methods for converting the hunspell and \TeX dictionaries and rule sets for use with open-source finite-state writer's tools. As concrete dictionaries we use the repositories of free implementations of these dictionaries and rule sets found on the internet, e.g. for the hunspell dictionary files found on the OpenOffice.org spell-checking site⁵. For hyphenation, we use the \TeX hyphenation patterns found on the \TeX hyphen page⁶.

In this section we describe the parts of the file formats we are working with. All of the information of the hunspell format specifics is derived from the `hunspell(4)`⁷ man page, as that is the only normative documentation of hunspell we have been able to locate. For \TeX hyphenation patterns, the reference documentation is Frank Liang's doctoral thesis [1] and the \TeX book [7].

A. Hunspell File Format

A hunspell spell-checking dictionary consists of two files: a dictionary file and an affix file. The dictionary file contains only root forms of words with information about morphological affix classes to combine with the roots. The affix file contains lists of affixes along with their context restrictions and effects, but the affix file also serves as a settings file for the dictionary, containing all meta-data and settings as well.

The dictionary file starts with a number that is intended to be the number of lines of root forms in the dictionary file, but in practice many of the files have numbers different from the actual line count, so it is safer to just treat it as a rough estimate. Following the initial line is a list of strings containing the root forms of the words in the morphology. Each word may be associated with an arbitrary number of classes separated by a slash. The classes are encoded in one of the three formats shown in the examples of Figure 1: a list of binary octets specifying classes from 1–255 (minus octets for CR, LF etc.), as in the Swedish example on lines 2–4, a

⁵<http://wiki.services.openoffice.org/wiki/Dictionaries>

⁶<http://www.tug.org/tex-hyphen/>

⁷<http://manpages.ubuntu.com/manpages/dapper/man4/hunspell.4.html>

```

1 # Swedish
  abakus/HDY
3 abalienation/AHDvY
  abalienera/MY
5 # Northern Sámi
  okta/1
7 guokte/1,3
  golbma/1,3
9 # Hungarian
  üzér/1 1
11 üzletág/2 2
   üzletvezető/3 1
13 üzletszerző/4 1

```

Fig. 1. Excerpts of Swedish, Northern Sámi and Hungarian dictionaries

list of binary words, specifying classes from 1–65,535 (again ignoring octets with CR and LF) or a comma separated list of numbers written in digits specifying classes 1–65,535 as in the North Sámi examples on lines 6–8. We refer to all of these as continuation classes encoded by their numeric decimal values, e.g. 'abakus' on line 2 would have continuation classes 72, 68 and 89 (the decimal values of the ASCII code points for H, D and Y respectively). In the Hungarian example, you can see the affix compression scheme, which refers to the line numbers in the affix file containing the continuation class listings, i.e. the part following the slash character in the previous two examples. The lines of the Hungarian dictionary also contain some extra numeric values separated by a tab which refer to the morphology compression scheme that is also mentioned in the affix definition file; this is used in the hunmorph morphological analyzer functionality which is not implemented nor described in this paper.

The second file in the hunspell dictionaries is the affix file, containing all the settings for the dictionary, and all non-root morphemes. The Figure 2 shows parts of the Hungarian affix file that we use for describing different setting types. The settings are typically given on a single line composed of the setting name in capitals, a space and the setting values, like the NAME setting on line 6. The hunspell files have some values encoded in UTF-8, some in the ISO 8859 encoding, and some using both binary and ASCII data at the same time. Note that in the examples in this article, we have transcribed everything into UTF-8 format or the nearest relevant encoded character with a displayable code point.

The settings we have used for building the spell-checking automata can be roughly divided into the following four categories: meta-data, error correction models, special continuation classes, and the actual affixes. An excerpt of the parts that we use in the Hungarian affix file is given in Figure 2.

The meta-data section contains, e.g., the name of the dictionary on line 6, the character set encoding on line 8, and the type of parsing used for continuation classes, which is omitted from the Hungarian lexicon indicating 8-bit binary parsing.

The error model settings each contain a small part of the

actual error model, such as the characters to be used for edit distance, their weights, confusion sets and phonetic confusion sets. The list of word characters in order of popularity, as seen on line 12 of Figure 2, is used for the edit distance model. The keyboard layout, i.e. neighboring key sets, is specified for the substitution error model on line 10. Each set of the characters, separated by vertical bars, is regarded as a possible slip-of-the-finger typing error. The ordered confusion set of possible spelling error pairs is given on lines 19–27, where each line is a pair of a 'mistyped' and a 'corrected' word separated by whitespace.

The compounding model is defined by special continuation classes, i.e. some of the continuation classes in the dictionary or affix file may not lead to affixes, but are defined in the compounding section of the settings in the affix file. In Figure 2, the compounding rules are specified on lines 14–16. The flags in these settings are the same as in the affix definitions, so the words in class 118 (corresponding to lower case v) would be eligible as compound initial words, the words with class 120 (lower case x) occur at the end of a compound, and words with 117 only occur within a compound. Similarly, special flags are given to word forms needing affixes that are used only for spell checking but not for the suggestion mechanism, etc.

The actual affixes are defined in three different parts of the file: the compression scheme part on the lines 1–4, the suffix definitions on the lines 30–33, and the prefix definitions on the lines 35–37.

The compression scheme is a grouping of frequently co-occurring continuation classes. This is done by having the first AF line list a set of continuation classes which are referred to as the continuation class 1 in the dictionary, the second line is referred to the continuation class 2, and so forth. This means that for example continuation class 1 in the Hungarian dictionary refers to the classes on line 2 starting from 86 (V) and ending with 108 (I).

The prefix and suffix definitions use the same structure. The prefixes define the left-hand side context and deletions of a dictionary entry whereas the suffixes deal with the right-hand side. The first line of an affix set contains the class name, a boolean value defining whether the affix participates in the prefix-suffix combinatorics and the count of the number of morphemes in the continuation class, e.g. the line 35 defines the prefix continuation class attaching to morphemes of class 114 (r) and it combines with other affixes as defined by the Y instead of N in the third field. The following lines describe the prefix morphemes as triplets of removal, addition and context descriptions, e.g., the line 31 defines removal of 'ö', addition of 'ös' with continuation classes from AF line 1108, in case the previous morpheme ends in 'ö'. The context description may also contain bracketed expressions for character classes or a fullstop indicating any character (i.e. a wild-card) as in the POSIX regular expressions, e.g. the context description on line 33 matches any Hungarian vowel except a, e or ö, and the 37 matches any context. The deletion and addition parts may also consist of a sole '0' meaning a zero-length string.

```

1 AF 1263
  AF VĚ-jxLnÓéè3ÄäTtYc,41 # 1
3 AF UmÖyiYcÇ # 2
  AF ÖCWRI-jpÓiyÉÁÿYc2 # 3
5
  NAME Magyar Ispell helyesírási szótár
7 LANG hu_HU
  SET UTF-8
9 KEY öüó|qwertyuiopőú| # wrap
  asdfghjkléáúíyxcvbnm
11 TRY íóútaeslzánorhgkié # wrap
  dmyőpvöbucfjüyxwq-á
13
  COMPOUNDBEGIN v
15 COMPOUNDEND x
  ONLYINCOMPOUND |
17 NEEDAFFIX u

19 REP 125
  REP í i
21 REP i í
  REP ó o
23 REP oliere oliére
  REP cc gysz
25 REP cs ts
  REP cs ds
27 REP ccs ts
  # 116 more REP lines

29
  SFX ? Y 3
31 SFX ? ö ős/1108 ö 20973
  SFX ? 0 ös/1108 [^aáeéííoóöüüü] 20973
33 SFX ? 0 s/1108 [áéííoóúúüüü-] 20973

35 PFX r Y 195
  PFX r 0 legújra/1262 . 22551
37 PFX r 0 legújjá/1262 . 22552
  # 193 more PFX r lines

```

Fig. 2. Excerpts from Hungarian affix file

As can be seen in the Hungarian example, the lines may also contain an additional number at the end which is used for the morphological analyzer functionalities.

B. \TeX Hyphenation Files

The \TeX hyphenation scheme is described in Frank Liang’s dissertation [1], which provides a packed suffix tree structure for storing the hyphenation patterns, which is a special optimized finite-state automaton. This paper merely reformulates the finite-state form of the patterns, for the purpose of obtaining a general finite-state transducer version of the rules to be combined with other pieces of the finite-state writer’s tools. In principle, the \TeX hyphenation files are like any \TeX source files, they may contain arbitrary \TeX code, and the only

```

\patterns {
2 .ach4
  .ad4der
4 .af1t
  .al3t
6 .am5at
  f5fin.
8 f2f5is
  f4fly
10 f2fy
}
\hyphenation {
12 as-so-ciate
14
  project
16 ta-ble
}

```

Fig. 3. Excerpts from English \TeX hyphenation patterns

requirement is that they have the ‘patterns’ command and/or the ‘hyphenation’ command. In practice, it is a convention that they do not contain anything else than these two commands, as well as a comment section describing the licensing and these conventions. The patterns section is a whitespace separated list of hyphenation pattern strings. The pattern strings are simple strings containing characters of the language as well as numbers marking hyphenation points, as shown in Figure 3. The odd numbers add a potential hyphenation point in the context specified by non-numeric characters, and the even numbers remove one, e.g. on line 8, the hyphen with left context ‘f’ and right context ‘fis’ would be removed, and a hyphen with left context ‘ff’ and right context ‘is’ is added. The numbers are applied in ascending order. The full-stop character is used to signify a word boundary so the rule on line 2 will apply to ‘ache’ but not to ‘headache’. The hyphenation command on lines 13–16 is just a list of words with all hyphenation points marked by hyphens. It has higher precedence than the rules and it is used for fixing mistakes made by the rule set.

IV. METHODS

This article presents methods for converting the existing spell-checking dictionaries with error models, as well as hyphenators to finite-state automata. As our toolkit we use the free open-source HFST toolkit⁸, which is a general purpose API for finite-state automata, and a set of tools for using legacy data, such as Xerox finite-state morphologies. For this reason this paper presents the algorithms as formulae such that they can be readily implemented using finite-state algebra and the basic HFST tools.

The lexc lexicon model is used by the tools for describing parts of the morphotactics. It is a simple right-linear grammar

⁸<http://HFST.sf.net>

for specifying finite-state automata described in [4], [8]. The twolc rule formalism is used for defining context-based rules with two-level automata and they are described in [9], [8].

This section presents both a pseudo-code presentation for the conversion algorithms, as well as excerpts of the final converted files from the material given in Figures 1, 2 and 3 of Section III. The converter code is available in the HFST SVN repository⁹, for those who wish to see the specifics of the implementation in lex, yacc, c and python.

A. Hunspell Dictionaries

The hunspell dictionaries are transformed into a finite-state transducer language model by a finite-state formulation consisting of two parts: a lexicon and one or more rule sets. The root and affix dictionaries are turned into finite-state lexicons in the lexc formalism. The Lexc formalism models the part of the morphotax concerning the root dictionary and the adjacent suffixes. The rest is encoded by injecting special symbols, called flag diacritics, into the morphemes restricting the morpheme co-occurrences by implicit rules that have been outlined in [10]; the flag diacritics are denoted in lexc by at-sign delimited substrings. The affix definitions in hunspell also define deletions and context restrictions which are turned into explicit two-level rules.

The pseudo-code for the conversion of hunspell files is provided in Algorithm 1 and excerpts from the conversion of the examples in Figures 1 and 2 can be found in Figure 4. The dictionary file of hunspell is almost identical to the lexc root lexicon, and the conversion is straightforward. This is expressed on lines 4–9 as simply going through all entries and adding them to the root lexicon, as in lines 6–10 of the example result. The handling of affixes is similar, with the exception of adding flag diacritics for co-occurrence restrictions along with the morphemes. This is shown on lines 10–28 of the pseudo-code, and applying it will create the lines 17–21 of the Swedish example, which does not contain further restrictions on suffixes.

To finalize the morpheme and compounding restrictions, the final lexicon in the lexc description must be a lexicon checking that all prefixes with forward requirements have their requiring flags turned off.

B. Hunspell Error Models

The hunspell dictionary configuration file, i.e. the affix file, contains several parts that need to be combined to achieve a similar error correction model as in the hunspell lexicon.

The error model part defined in the KEY section allows for one slip of the finger in any of the keyboard neighboring classes. This is implemented by creating a simple homogeneously weighted crossproduct of each class, as given on lines 1–7 of Algorithm 2. For the first part of the example on line 10 of Figure 2, this results in the lexc lexicon on lines 11–18 in Figure 5.

The error model part defined in the REP section is an arbitrarily long ordered confusion set. This is implemented

Algorithm 1 Extracting morphemes from hunspell dictionaries

```

finalflags ← ε
2: for all lines morpheme/Conts in dic do
    flags ← ε
4:   for all cont in Conts do
        flags ← flags + @C.cont@
6:   LexConts ← LexConts ∪ 0:[<cont] cont
    end for
8:   LexRoot ← LexRoot ∪ flags + morpheme Conts
    end for
10:  for all suffixes lex, deletions, morpheme/Conts, context
    in aff do
        flags ← ε
12:   for all cont in Conts do
            flags ← flags + @C.cont@
14:   LexConts ← LexConts ∪ 0 cont
        end for
16:   Lexlex ← Lexlex ∪ flags + [<lex] + morpheme Conts
        for all del in deletions do
18:           lc ← context + deletions before del
                rc ← deletions after del + [<lex] + morpheme
20:           Twol_d ← Twol_d ∩ del:0 ⇔ lc _ rc
        end for
22:   Twol_m ← Twol_m ∩ [<lex] : 0 ⇔ context _ morpheme
        end for
24:  for all prefixes lex, deletions, morpheme/conts, context
    in aff do
        flags ← @P.lex@
26:   finalflags ← finalflags + @D.lex@
        lex → prefixes {otherwise as with suffixes, swapping
        left and right}
28:  end for
    Lex_end ← Lex_end ∪ finalflags #

```

by simply encoding them as increasingly weighted paths, as shown in lines 9–12 of the pseudo-code in Algorithm 2.

The TRY section such as the one on line 12 of Figure 2, defines characters to be tried as the edit distance grows in descending order. For a more detailed formulation of a weighted edit distance transducer, see e.g. [2]). We created an edit distance model with the sum of the positions of the characters in the TRY string as the weight, which is defined on lines 14–21 of the pseudo-code in Algorithm 2. The initial part of the converted example is displayed on lines 20–27 of Figure 5.

Finally to attribute different likelihood to different parts of the error models we use different weight magnitudes on different types of errors, and to allow only correctly written substrings, we restrict the result by the root lexicon and morfotax lexicon, as given on lines 1–9 of Figure 5. With the weights on lines 1–5, we ensure that KEY errors are always suggested before REP errors and REP errors before TRY errors. Even though the error model allows only one error of any type, simulating the original hunspell, the resulting

⁹<http://hfst.svn.sourceforge.net/viewvc/hfst/trunk/conversion-scripts/>

```

LEXICON Root
2   HUNSPELL_pfx ;
   HUNPELL_dic ;
4
! swedish lexc
6 LEXICON HUNSPELL_dic
  @C.H@@C.D@@C.Y@abakus HDY ;
8  @C.A@@C.H@@C.D@@C.v@@C.Y@abalienation
   HUNSPELL_AHDvY ;
10 @C.M@@C.Y@abalienera MY ;

12 LEXICON HDY
   0:[<H]   H ;
14  0:[<D]   D ;
   0:[<Y]   Y ;
16

LEXICON H
18  er HUNSPELL_end ;
   ers HUNSPELL_end ;
20  er HUNSPELL_end ;
   ers HUNSPELL_end ;
22

LEXICON HUNSPELL_end
24  @D.H@@D.D@@D.Y@@D.A@@D.v@@D.m@ # ;

26 ! swedish twolc file
Rules
28 "Suffix H allowed contexts"
  %[%<H%]: 0 <=> \ a _ e r ;
30   \ a _ e r s ;
   a:0 _ e r ;
32   a:0 _ e r s ;

34 "a deletion contexts"
  a:0 <=> _ %[%<H%]:0 e r ;
36   _ %[%<H%]: e r s ;

```

Fig. 4. Converted dic and aff lexicons and rules governing the deletions

transducer can be transformed into an error model accepting multiple errors by a simple FST algebraic concatenative n-closure, i.e. repetition.

C. $\text{T}_{\text{E}}\text{X}$ Hyphenation

The formulation of hyphenation as finite-state transducers is simple. We use the hyphenation alphabet $\Sigma_H = -$. To model the context-based deletions and additions of hyphenation patterns, we use twol rules with centers of $\epsilon : -$ for addition and $- : \epsilon$ for deletion. The algorithm for creating the rule sets described in Algorithm 3 simply goes through the patterns, and for each hyphenation point of each pattern extracts left and right context strings and adds them to the contexts of a rule. The result is exemplified in Figure 6. There is one rule for each of the hyphenation point numbers. The rules may be composed into one single transducer at compile time or

Algorithm 2 Extracting patterns for hunspell error models

```

for all neighborsets  $ns$  in KEY do
2:   for all character  $c$  in  $ns$  do
       for all character  $d$  in  $ns$  such that  $c! = d$  do
4:      $Lex_{KEY} \leftarrow Lex_{KEY} \cup c : d_{<0>}\#$ 
       end for
6:   end for
end for
8:  $w \leftarrow 0$ 
   for all pairs  $wrong, right$  in REP do
10:   $w \leftarrow w + 1$ 
      $LEX_{REP} \leftarrow LEX_{REP} \cup wrong : right_{<w>}\#$ 
12: end for
    $w \leftarrow 0$ 
14: for all character  $c$  in TRY do
      $w \leftarrow w + 1$ 
16:   $Lex_{TRY} \leftarrow Lex_{TRY} \cup c : 0_{<w>}\#$ 
      $Lex_{TRY} \leftarrow Lex_{TRY} \cup 0 : c_{<w>}\#$ 
18:  for all character  $d$  in TRY such that  $c! = d$  do
      $Lex_{TRY} \leftarrow Lex_{TRY} \cup c : d_{<w>}\#$  {for swap: replace
     # with  $cd$  and add  $Lex_{cd} \cup d : c_{<0>}\#$ }
20:  end for
end for

```

Algorithm 3 Extracting hyphenation patterns from $\text{T}_{\text{E}}\text{X}$

```

for all patterns  $p$  do
2:   for all digits  $d$  in  $p$  do
      $l, r \leftarrow \text{split } p \text{ on } d$ 
4:     if  $d$  odd then
        $l, r \leftarrow l, r << 0 : \epsilon$ 
6:        $Twol_d \leftarrow Twol_d \cap 0 : \epsilon \leftrightarrow l_r$ ;
     else
7:        $l, r \leftarrow l, r << \epsilon : 0$ 
8:        $Twol_d \leftarrow Twol_d \cap \epsilon : 0 \leftrightarrow l_r$ ;
10:    end if
    end for
12:  end for
   for all hyphenations  $h$  do
14:     $word \leftarrow h - \text{hyphens}$ 
      $Lex_{exceptions} \leftarrow Lex_{exceptions} \cup word : h \#$ 
16:  end for

```

applied as cascade at runtime.

The $\text{T}_{\text{E}}\text{X}$ hyphenation pattern also contains explicit exceptions to the hyphenation patterns, which are simply specific word forms with hyphenations, and can be compiled as simple paths: e.g. for the pattern{as-so-ciate} we create a path $asesoeciate : as - so - ciate$

V. IMPLEMENTATION AND TESTS

We have implemented the spell-checkers and hyphenators as finite-state transducers using program code and scripts with a Makefile. To test the code, we have converted 49 hyphenation pattern files and more than 42 hunspell dictionaries from various language families. They consist of the dictionaries that

```

LEXICON HUNSPELL_error_root
2 < ? > HUNSPELL_error_root ;
  HUNSPELL_KEY "weight: 0" ;
4 HUNSPELL_REP "weight: 100" ;
  HUNSPELL_TRY "weight: 1000" ;
6
LEXICON HUNSPELL_errret
8 < ? > HUNSPELL_errret ;
  # ;
10
LEXICON HUNSPELL_KEY
12 ö:ü HUNSPELL_errret "weight: 0" ;
  ö:ó HUNSPELL_errret "weight: 0" ;
14 ü:ö HUNSPELL_errret "weight: 0" ;
  ü:ó HUNSPELL_errret "weight: 0" ;
16 ó:ö HUNSPELL_errret "weight: 0" ;
  ó:ü HUNSPELL_errret "weight: 0" ;
18 ! same for other parts
20
LEXICON HUNSPELL_TRY
  í:0 HUNSPELL_errret "weight: 1" ;
22 0:í HUNSPELL_errret "weight: 1" ;
  í:ó HUNSPELL_errret "weight: 2" ;
24 ó:í HUNSPELL_errret "weight: 2" ;
  ó:0 HUNSPELL_errret "weight: 2" ;
26 0:ó HUNSPELL_errret "weight: 2" ;
  ! same for rest of the alphabet
28
LEXICON HUNSPELL_REP
30 í:i HUNSPELL_errret "weight: 1" ;
  i:í HUNSPELL_errret "weight: 2" ;
32 ó:o HUNSPELL_errret "weight: 3" ;
  oliere:olière HUNSPELL_errret "weight: 4" ;
34 cc:gysz HUNSPELL_errret "weight: 5" ;
  cs:ts HUNSPELL_errret "weight: 6" ;
36 cs:ds HUNSPELL_errret "weight: 7" ;
  ccs:ts HUNSPELL_errret "weight: 8" ;
38 ! same for rest of REP pairs ...

```

Fig. 5. Converted error models from aff file

```

1 "Hyphen insertion 1"
0:%- <=> # (0:%-) a (0:%-) f _ t ;
3 ...

```

Fig. 6. Converted hyphenation models from \TeX examples

were accessible from the aforementioned web sites at the time of writing. The Tables I and II gives an overview of the sizes of the compiled automata. The size is given in binary multiples of bytes as reported by `ls -hl`.

In the hyphenation table, the second column gives the number of patterns in the rules. The total size is a result of composing all hyphenation rules into one transducer; it may be noted that both separate rules and a single composed transducer are equally usable at runtime. The separated version requires less memory whereas the single composed version is faster. For large results, such as the Norwegian¹⁰ one, it may still be beneficial to keep the rules separated. In the Norwegian case, the four separately compiled rules are each of sizes between 1.2 MiB and 9.7 MiB.

For the hunspell automata in Table II, we also give the number of roots in the dictionary file and the affixes in affix file. These numbers should also help with identifying the version of the dictionary, since there are multiple different versions available in the downloads.

The resulting transducers were tested by hand using the results of the corresponding \TeX `hyphenate` command and `hunspell -d` as well as the authors' language skills to judge errors. As testing material, a wikipedia article on the Finnish language¹¹ were used for most languages, and some arbitrary articles where this particular article was not found. In both tests, the majority of differences come from the lack of normalization or case folding. E.g. this resulted in our converted transducers failing to hyphenate words where uppercase letters would have been equal to their lowercase variants.

The hunspell model was built incrementally starting from the basic set of affixes and dictionary, and either adding or skipping all directive types of the file format as found in the wild. Some of the omissions show up e.g. in the English results, where omitting of the PHONE directive for the suggestion mechanism results in some of the differing suggestions in English tests, e.g. first suggestion for *calqued* in hunspell is *catafalqued*. Without implementing the phonetic folding, we get no results within 1 hunspell error, and get word forms like *chalked*, *caulked*, and so forth, within 2 hunspell errors. No other language has .aff files with PHONE rules, e.g. in French *comitatif* gets the suggestions *commutatif* and *limitatif* as the first ones in both systems.

VI. CONCLUSION

We have demonstrated a method and created the software to convert legacy spell-checker and hyphenation data to a more general framework of finite-state automata and used it in a real-life application. We have also referred to methods for extending the system to more advanced error models and the inclusion of other more complex models in the same system. We are currently developing a platform for finite-state based spell-checkers for open-source systems in order to improve the front-end internationalization.

¹⁰both Nynorsk and bokmål input the same patterns

¹¹<http://en.wikipedia.org/wiki/Finnish+language> and its international links

TABLE I
COMPILED HYPHENATION AUTOMATA SIZES

Language	Hyphenator total	Number of patterns
Norwegian	978 MiB	27,166
German (Germany, 1996)	72 MiB	14,528
German (Germany, 1901)	66 MiB	14,323
Dutch	58 MiB	12,742
English (Great Britain)	38 MiB	8,536
Irish	20 MiB	6,046
English (U.S.)	19 MiB	4,948
Hungarian	15 MiB	13,469
Swedish	12 MiB	4,717
Icelandic	12 MiB	4,199
Estonian	8.8 MiB	3,701
Russian	4.2 MiB	4,820
Czech	3.1 MiB	3,646
Ancient Greek	2.4 MiB	2,005
Ukrainian	1.5 MiB	1,269
Danish	1.4 MiB	1,153
Slovak	1.1 MiB	2,483
Slovenian	939 KiB	1,086
Spanish	546 KiB	971
French	521 KiB	1,184
Interlingua	382 KiB	650
Greek (Polyton)	325 KiB	798
Upper Sorbian	208 KiB	1,524
Galician	160 KiB	607
Romanian	151 KiB	665
Mongolian	135 KiB	532
Finnish	111 KiB	280
Catalan	95 KiB	231
Greek (Monoton)	91 KiB	429
Serbian	76 KiB	2,681
Serbocroatian	56 KiB	2,681
Sanskrit	32 KiB	550
Croatian	32 KiB	1,483
Coptic	30 KiB	128
Latin	26 KiB	87
Bulgarian	24 KiB	1,518
Portuguese	19 KiB	320
Basque	15 KiB	49
Indonesian	14 KiB	46
Turkish	8 KiB	602
Chinese (Pinyin)	868	202

TABLE II
COMPILED HUNSPELL AUTOMATA SIZES

Language	Dictionary	Roots	Affixes
Portugese (Brazil)	14 MiB	307,199	25,434
Polish	14 MiB	277,964	6,909
Czech	12 MiB	302,542	2,492
Hungarian	9.7 MiB	86,230	22,991
Northern Sámi	8.1 MiB	527,474	370,982
Slovak	7.1 MiB	175,465	2,223
Dutch	6.7 MiB	158,874	90
Gascon	5.1 MiB	2,098,768	110
Afrikaans	5.0 MiB	125,473	48
Icelandic	5.0 MiB	222,087	0
Greek	4.3 MiB	574,961	126
Italian	3.8 MiB	95,194	2,687
Gujarati	3.7 MiB	168,956	0
Lithuanian	3.6 MiB	95,944	4,024
English (Great Britain)	3.5 MiB	46,304	1,011
German	3.3 MiB	70,862	348
Croatian	3.3 MiB	215,917	64
Spanish	3.2 MiB	76,441	6,773
Catalan	3.2 MiB	94,868	996
Slovenian	2.9 MiB	246,857	484
Faeroese	2.8 MiB	108,632	0
French	2.8 MiB	91,582	507
Swedish	2.5 MiB	64,475	330
English (U.S.)	2.5 MiB	62,135	41
Estonian	2.4 MiB	282,174	9,242
Portugese (Portugal)	2 MiB	40,811	913
Irish	1.8 MiB	91,106	240
Friulian	1.7 MiB	36,321	664
Nepalese	1.7 MiB	39,925	502
Thai	1.7 MiB	38,870	0
Esperanto	1.5 MiB	19,343	2,338
Hebrew	1.4 MiB	329,237	0
Bengali	1.3 MiB	110,751	0
Frisian	1.2 MiB	24,973	73
Interlingua	1.1 MiB	268,50	54
Persian	791 KiB	332,555	0
Indonesian	765 KiB	23,419	17
Azerbaijani	489 KiB	19,132	0
Hindi	484 KiB	15,991	0
Amharic	333 KiB	13,741	4
Chichewa	209 KiB	5,779	0
Kashubian	191 KiB	5,111	0

The next obvious development for the finite-state spell checkers is to apply the unigram training [2] to the automata, and extend the unigram training to cover longer n-grams and real word error correction.

ACKNOWLEDGMENT

The authors would like to thank Miikka Silfverberg and others in the HFST research team as well as the anonymous reviewers for useful comments on the manuscript.

REFERENCES

- [1] F. M. Liang, "Word hy-phen-a-tion by com-pu-ter," Ph.D. dissertation, Stanford University, 1983. [Online]. Available: <http://www.tug.org/docs/liang/>
- [2] T. A. Pirinen and K. Lindén, "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 languages*, Valletta, Malta, 2010, pp. 13–18. [Online]. Available: http://siuc01.si.edu/~jipsagak/SALTMIL2010_Proceedings.pdf
- [3] L. A. Wilcox-O’Hearn, G. Hirst, and A. Budanitsky, "Real-word spelling correction with trigrams: A reconsideration of the mays, damerau, and mercer model," in *CICLing*, ser. Lecture Notes in Computer Science, A. F. Gelbukh, Ed., vol. 4919. Springer, 2008, pp. 605–616.
- [4] K. R. Beesley and L. Karttunen, *Finite State Morphology*. CSLI publications, 2003.
- [5] M. Silfverberg and K. Lindén, "Hfst runtime format—a compacted transducer format allowing for fast lookup," in *FSMNLP 2009*, B. Watson, D. Courie, L. Cleophas, and P. Rautenbach, Eds., 13 July 2009. [Online]. Available: <http://www.ling.helsinki.fi/~klinden/pubs/fsmnlp2009runtime.pdf>
- [6] M. Mohri and M. Riley, "An efficient algorithm for the n-best-strings problem," 2002.
- [7] D. Knuth, *The TeXbook*. Oxford Oxfordshire: Oxford University Press, 1986.
- [8] K. Lindén, M. Silfverberg, and T. Pirinen, "Hfst tools for morphology—an efficient open-source package for construction of morphological analyzers," in *sfcM 2009*, ser. Lecture Notes in Computer Science, C. Mahlow and M. Piotrowski, Eds., vol. 41. Springer, 2009, pp. 28–47.
- [9] K. Koskenniemi, "Two-level morphology: A general computational model for word-form recognition and production," Ph.D. dissertation, University of Helsinki, 1983. [Online]. Available: <http://www.ling.helsinki.fi/~koskenni/doc/Two-LevelMorphology.pdf>
- [10] K. R. Beesley, "Constraining separated morphotactic dependencies in finite-state grammars." Morristown, NJ, USA: Association for Computational Linguistics, 1998, pp. 118–127.