UNIVERSITY OF MURCIA

International Journal of English Studies

IJES

www.um.es/ijes

The Finite-State Playground

MICHAEL HAMMOND
*University of Arizona**

ABSTRACT

Finite-state methods are finding ever increasing use among linguists as a way of modeling phonology and morphology and as a method for manipulating and modeling text. This paper describes a suite of very simple finite-state tools written by the author that can be used to investigate this area and that can be used for simple analysis.

KEYWORDS: Finite-state, automata, toolkits, phonology, morphology.

**Address for correspondence*: Michael Hammond, Department of Linguistics, University of Arizona. Douglass Buiding, 3rd floor. Tucson, Arizona, USA. Tel: (520) 621-5759. E-mail: hammond@u.arizona.edu

I. INTRODUCTION

This paper describes and exemplifies the *finite-state playground* (FSP), a set of simple command-line tools for building and manipulating finite-state models of language. There are eight programs in the suite that can create finite automata from regular expressions or text. The automata created can be manipulated in various ways, e.g. minimized, determinized, reversed, etc. In addition, all the usual finite-state closure operations are supported: union, concatenation, Kleene star, and intersection. There is also limited support for weighted automata. Finally, automata can be saved in an XML-based format or output in a form suitable for display with the free Graphviz package.[1.] Most of the programs can take XML-encoded automata as input and this provides a mechanism for incrementally building up large-scale models.

In this paper, we first outline the logic of finite-state descriptions of natural language. We then describe the basic usage of the software and give some simple examples. In addition, we compare this package to some of the other finite-state software packages available. Finally, we describe some of challenges of writing software for linguistic applications.

II. FINITE-STATE MODELING

In this section, we outline the basic structure of finite-state automata (FSAs) and some of the algorithms for FSAs implemented in the software.

Finite-state automata have a long history and very well-understood mathematical properties. The broader class of Turing machines was originally proposed in Turing (1936). The general logic of finite automata per se was developed in McCulloch & Pitts (1943), and then fleshed out mathematically in Kleene (1956). Hopcroft & Ullman (1979), Lawson (2004) and many others provide overviews.

There is also a long tradition of finite-state approaches to linguistics. This is most pronounced in the area of phonology. The claim that phonology could be treated in finite-state terms was first advanced in Johnson (1972) and then in Kaplan & Kay (1994). Karttunen (1983) was also very influential in showing how finite-state approaches could describe phonological and morphological patterning. Bird & Ellison (1994) and Bird (1995) provide accounts of nonlinear phonology and finite-state approaches to Optimality Theory include Ellison (1994) and Karttunen (1983).

The basic idea is quite simple. An FSA can be seen as a very primitive model of a machine which reads a string of symbols. As it reads each symbol, it moves to a new state. If, at the end of the string of symbols, the machine is in a special *final* state, the string is said to be accepted. If, on the other hand, the machine is not in one of those states, or no move was available at any point, then the string is not accepted.

A simple example is given in Figure 1. The machine begins in state q_0. It reads any number of instances of the symbol *a*. It can then either read one or more instances of *b* followed by a single *c*, or it can read a single *d*. Either of those results in the machine terminating in state q_2.

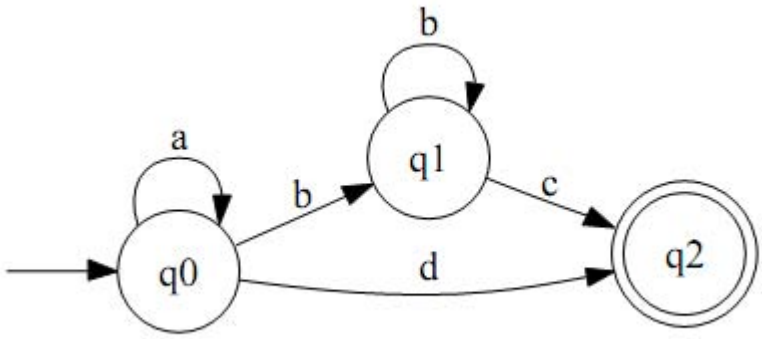

Figure 1: A simple automaton

Consider how this would work with the string *abbc*. The machine begins in state q_0 and reads the first symbol *a*. This moves the machine back to q_0. The second symbol, *b*, is read moving the machine to state q_1. The third symbol, *b*, is read and the machine returns to state q_1. Finally, the last symbol *c* is read moving the machine to q_2. The latter is a designated final state, indicated with the double circle, and so the string is accepted.

Consider now how the machine would treat *ac*. First, the symbol *a* is read moving the machine from q_0 back to q_0. The symbol *c* is then read, but there is no arc labeled *c* from q_0. Hence, the string *ac* is not accepted.

FSAs can be more complex and can be used to represent phonological and morphological generalizations. For example, the automaton in Figure 2 represents the schematized syllabification possibilities for a word for a simplified version of English.[2.] Here, syllables can be of the form (C)(C)V(C)(C) medially. An extra consonant is allowed at each edge of the word. All words contain at least one syllable.

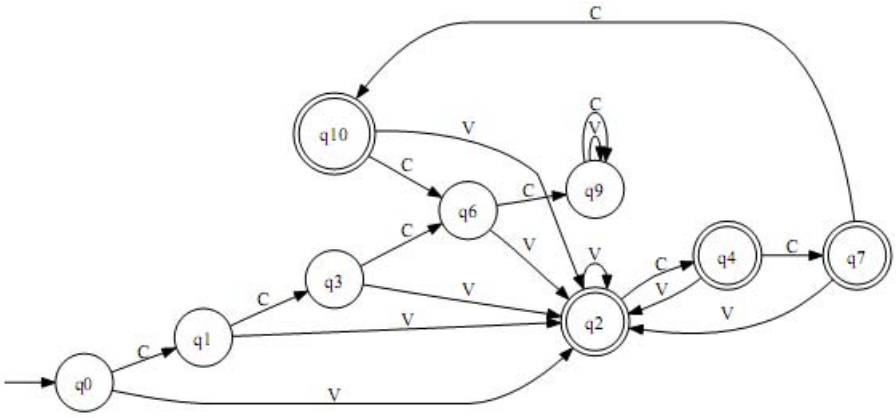

Figure 2: A simple syllable template

The automaton in Figure 3 provides a simple schematized morphological example. The phenomenon in question is recursive affixation in English. A word like *nation* can get suffixed with *–al* producing *national*. This, in turn, can be suffixed with *–ize*, producing *nationalize*. That can undergo suffixation again with *–ation* producing *nationalization*. Arguably, this whole process can proceed ad infinitum, as in (1).

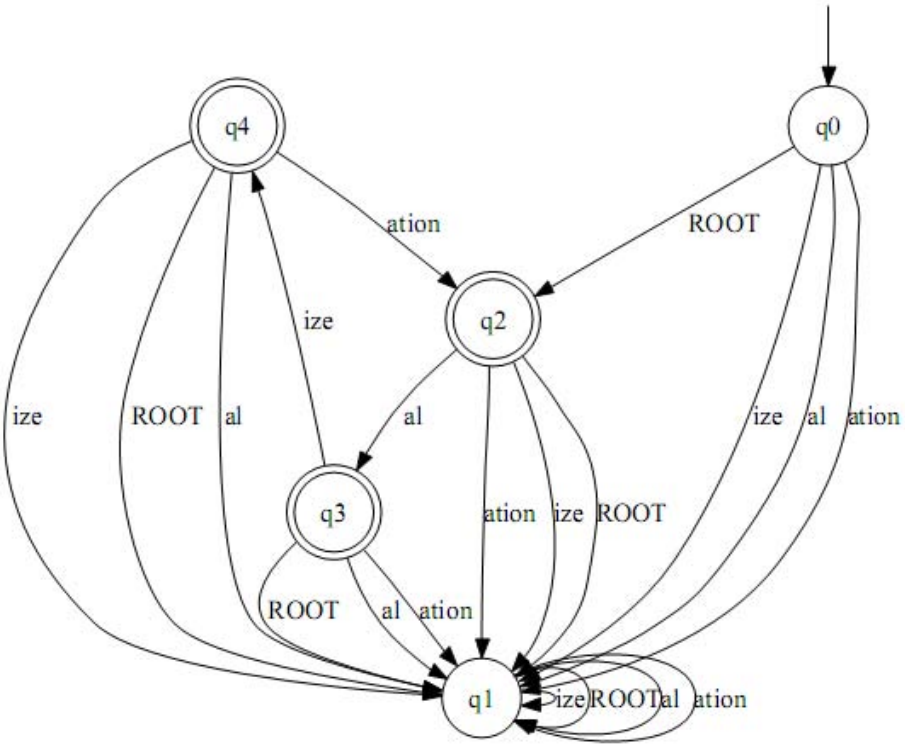

Figure 3: A simple morphological template

(1) nation+al+iz+ation+al+iz+ation+…

FSAs like these can be used to model, understand, and test the phonological and morphological well-formedness of words. We will show how these sorts of models can be constructed by the FSP package in Section IV below.

Standard references like Hopcroft & Ullman (1979) give a formal characterization, but in the remainder of this section we provide an informal characterization of what an FSA is capable of.

As described above, FSAs will accept or reject different input strings. Which strings are accepted is a consequence of how the automaton is constructed. There are really only three things that an FSA can do: concatenation (put two symbol sequences in sequence), union (choose between two different symbol sequences), or Kleene star (repeat some symbol sequence). Let's look at each of these.

Putting several arcs and states in sequence allows the FSA to accept strings that have the corresponding symbols in sequence. Thus, the partial FSA in Figure 4 accepts strings composed of …*ab*….

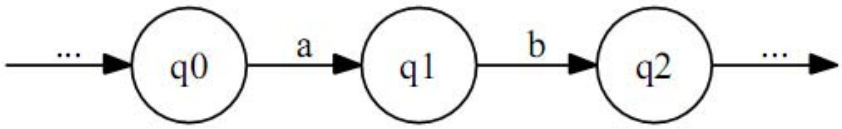

Figure 4: Concatenation with an FSA

Putting two separate arcs from a single state allows the FSA to accept one or the other string. Thus the partial FSA in Figure 5 accepts strings composed of …*a*… or …*b*….

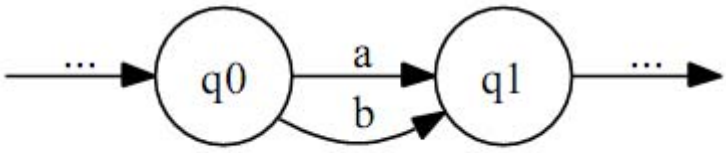

Figure 5: Union with an FSA

Finally, allowing an arc to loop back to a previous state, either directly or indirectly, allows the string of symbols along the loop to be repeated. For example, the partial FSA in Figure 6 accepts any number of repetitions of the substring *ab*, e.g. …, …*ab*…, …*abab*…, etc.

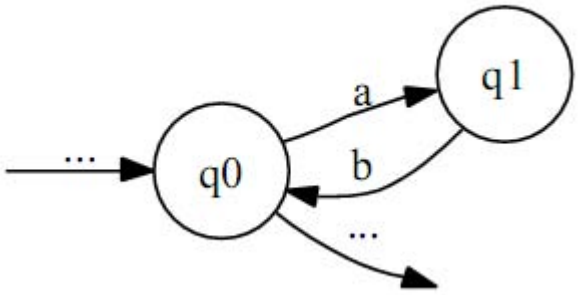

Figure 6: Looping with an FSA

As long as a language can be described using these three operations—concatenation, union, and Kleene star—it can be described with an FSA. This corresponds to the long-established equivalence of finite-state automata and *regular expressions*. The latter are all and only the languages that can be described with these three operations. For example, the regular expression $a(b|c^*)$ exemplifies all three operations and represents the language where all strings are composed of a single instance of *a* followed by precisely one *b* or followed by any number of instances of *c*, e.g. *ab*, *a*, *ac*, *acc*, *accc*, etc. Concatenation is indicated by linear precedence. Kleene star is indicated by an asterisk. Union is indicated by the tie-bar symbol and grouping of symbols is indicated when necessary with parentheses. An automaton that accepts this language is given in Figure 7.

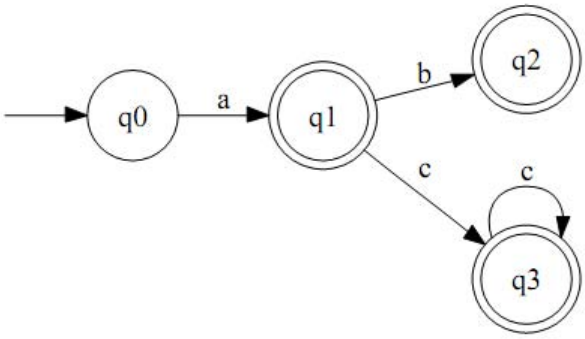

Figure 7: All three operations in an FSA

The upshot is that, while certain kinds of languages can be defined in terms of FSAs, certain kinds of languages cannot. A very simple example is $(ab)^*$ vs. $a^n b^n$. The first describes the language where all strings are composed of any number of repetitions of the sequence *ab*, e.g. *ab*, *abab*, *ababab*, etc. This language falls within the realm of regular expressions and can be described by an automaton that is very similar to that in Figure 6. The second pattern describes the language where every string is composed of some number of instances of *a* followed by the same number of instances of *b*, e.g. *ab*, *aabb*, *aaabbb*, etc. This is not a regular expression, since superscripts go beyond concatenation, union, and Kleene star. Such a language cannot be described with an FSA. It is widely believed that

while phonology and morphology can be described with finite automata, syntax cannot be fully described with them.

There are a number of operations and corresponding algorithms that are defined for FSAs:

Concatenation Two separate FSAs can be concatenated into a new FSA.

Union One can construct the union of two separate FSAs.

Kleene star One can create an FSA that represents some other FSA repeated any number of times.

Determinization Automata can be determinized so that for any string that is considered there is one and only one path through the FSA.

Minimization FSAs can be minimized; there is an algorithm that is guaranteed to find the FSA with the smallest number of states that can be used to describe a regular language.

Complementation If one FSA describes some particular language *L* over some alphabet Σ, then one can create an FSA that accepts any string over Σ that is *not* in *L*.

Intersection Given two FSAs that describe two distinct languages, one can construct a single new FSA that allows only those strings that are accepted by each independent FSA.

Reversal One can construct an FSA that accepts strings that are reversed from some other FSA.

These are very powerful and convenient tools and the FSP software implements all of them. These allow one to readily construct FSAs for a variety of analysis tasks.

III. WHAT THE SOFTWARE DOES

The software is actually a set of separate command-line programs that allow one to create, combine, and manipulate FSAs in a variety of ways. In this section, we describe in general terms what each program does and how they can be combined.

The centerpiece of the set is the re program. The basic logic of the program is that an automaton can be created from a regular expression or read in from a stored FSA. Various manipulations of the automaton can be made and the FSA can then be saved in one of several formats. The different options are controlled with the program flags listed in Figure 8.

-i	filename	read FSA from XML file
-r	'regexp'	parse regular expression
-f		create FSA from regular expression
-e		remove epsilon-arcs from FSA
-d		determinize FSA
-p		print FSA (Graphviz format)
-w		include weights with -p
-x		save FSA as XML
-s	string	test string against FSA
-m		minimize FSA
-M		minimize FSA (Brzozowski)
-c		take complement of FSA
-R		reverse FSA

Figure 8: Flags for re program

If there is no argument, then the flags can be combined in a block with a single preceding hyphen, e.g. -fedMx or with separate hyphens, e.g. -f -e -d -M -x. For the flags with arguments, the argument must immediately follow that flag, as exemplified below.

We might create an FSA for the regular expression *a*(*b*|*c**) with the following code:

(2) re -fr 'a(b|c*)'

Here the -r flag indicates that the following argument is a regular expression. The -f flag indicates that we are to create an FSA. If we had a stored automaton in the file fsa1.xml, we could read it into the program with the following code.

(3) re -i fsa1.xml

Automata can be saved in either of two formats. One possibility is the dot format. This allows us to use the Graphviz package to create a graphical rendering of the FSA.[3.] If we wanted to create a file fsa1.dot for this purpose, we would use either of the statements below, depending on how we had created the FSA to begin with.

(4)
```
re -fpr 'a(b|c*)' > fsa1.dot
re -pi fsa1.xml > fsa1.dot
```

Here we've used the > symbol to redirect the output of the re program into a file. That file can then be opened with the relevant Graphviz program for rendering.[4.]

We can also save FSAs in an XML-based format. This is useful if we want to incrementally create an automaton, each time saving our stepwise results in a file and then using that file to make the next manipulation. The following shows how we can do this for the two scenarios we've exemplified, saving the resulting automaton in a file fsa2.xml.

(5)
```
re -fxr 'a(b|c*)' > fsa2.xml
re -xi fsa1.xml > fsa2.xml
```

Notice that we've simply changed the -p flag to the -x flag and redirected the output to a different filename.

Most of the flags given in Figure 8 are self-explanatory, but several require some explanation. The -e flag removes arcs labeled with an epsilon. These are actually arcs that the FSA can follow without reading a symbol. It turns out that a number of the algorithms are easiest to implement if we allow the algorithm to produce an FSA with ∈-arcs. Such automata are not any more powerful than automata without such arcs. Nonetheless for subsequent manipulation, it is usually best to then remove those arcs. This flag does that.

The -m and -M flags allow for two different algorithms for minimizing an FSA. The first is the standard algorithm (Hopcroft & Ullman, 1979); the latter is due to Brzozowski (1962).

The programs include limited support for adding weights to arcs. The weights reflect the relative likelihood that some arc might be followed rather than some other. We will return to this in the discussion of the train program below. For now, the -w flag includes weights when an FSA is output for display.

Finally, the -s flag allows one to test whether some string is well-formed with respect to some automaton. The automaton must be determinized for this to have an effect. The following command creates an automaton for the regular expression *a*(*b*|*c**). It removes □-arcs and determinizes the FSA with the -e and -d flags. It then tests the resulting FSA against the string *abc* with the -s flag. The last is not accepted by the automaton and the program reports that result.

(6) re -fedr 'a(b | c*)' -s 'abc'

Running a legal string against the FSA will report that it is, in fact, a legal string. For example:

(7) re -fedr 'a(b | c*)' -s 'accc'

Notice that, as in the previous case, some combinations of flags have no effect. For example, weights are always included in XML output, so the -w flag has no effect there. On the other hand, some combinations are required. As we just saw, when an FSA is created using the -f flag, then it must have n-arcs removed and be determinized with the -de combination for the -s flag to have an effect.

Let's now consider some of the other programs in the FSP suite. Four of them are quite straightforward; they perform various closure operations on FSAs that are input in XML format.

The concatenate program takes two FSAs and produces a new FSA that concatenates them. For example, if we create an FSA based on the regular expression *a*(*b*|*c*) and save it as abc.xml, and we create another FSA based on the regular expression *de** and save it as de.xml, then we can use concatenate to produce their concatenation as follows:

(8) concatenate abc.xml de.xml>abcde.xml

The new FSA will accept *a*(*b*|*c*)*de**.

The kleene program constructs the Kleene closure of another automaton. For example, if abc.xml encodes *a*(*b*|*c*), then we can run the kleene program on it as follows:

(9) kleene abc.xml>abcK.xml

The new FSA accepts (*a*(*b*|*c*))*.

The union program produces the union of two FSAs. For example, if abccc.xml accepts *abc** and aaabc.xml accepts *a*bc*, then we can use union as follows:

(10) union abccc.xml aaabc.xml>aabcc.xml

The new automaton accepts (*abc**|*a*bc*).

Finally, the intersect program produces the intersection of two FSAs. Using the same examples as in the previous case, we execute the program as follows:

(11) intersect abccc.xml aaabc.xml>abc.xml

The new automaton accepts only *abc*, the intersection of *abc** and *a*bc*.

The dictionary program goes beyond simple FSA operations. Specifically, it produces an FSA from a list of words or strings. The latter are given in a text file where each string is on a separate line. The resulting automaton is output in XML format. For example, we can create a text file words.txt with the following content:

(12) hat,lemon,oranges

The following command will produce an FSA that accepts just these words.

(13) dictionary words.txt>words.xml

Finally, the last two programs in the suite provide limited support for adding and manipulating weights.

The train program takes a determinized automaton and a list of strings and increments the weights in the automaton accordingly: each time an arc is used for a string, the count for that arc is increased by one. Imagine we first create an FSA using the dictionary program as above. We then create a training file train.txt as follows:

(14) hat,lemon,hat

We would invoke the train program as follows:

(15) train words.xml train.txt>trained.xml

The new FSA has exactly the same structure, except that the arcs that apply to lemon would have a weight of 1 added to each of them and the arcs associated with hat would have weights of 2 added. The arcs associated only with oranges would still have 0 associated with them.

The last program in the suite removes arcs from an FSA if the weight of the arc falls below a specified threshold. For example, to remove the arcs associated only with oranges from the FSA we just trained, we would execute this command:

(16) winnow trained.xml 1>pruned.xml

To do the same for the arcs associated with lemon, we would specify a higher threshold:

(17) winnow trained.xml 2>pruned.xml

The new automaton would accept only hat.

Support for weights is clearly incomplete. Other than the re program outputting weights with the -p flag, there are only the facilities just exemplified.

The full suite of programs is quite minimal. The idea is that each invocation of a program implements known algorithms. Complex operations can be achieved only by successive application of different programs where each stage is saved as an XML file.[5.]

While this user model is certainly less convenient, it provides a great deal of room to understand precisely what's going on with different operations. In addition, it allows for great flexibility in crafting complex sequences of program applications, perhaps automated in a shell script or batch file.

Additionally, the programs make use of several external technologies. The Graphviz package is used for graphics. In addition, XML is the format for saving and reusing automata. The latter allows for the possibility of extending the package using external XML-based applications in the future.

IV. EXAMPLES

In this section, we show how to use the package to create and test the simple examples presented in Figure 2 and in Figure 3.

Let's consider the syllabification example first. This example represents the generalization that words in some language are composed of one or more syllables. The syllable has the general form (C)(C)V(C)(C), and this can be repeated any number of times in a word. In addition, the language allows an extra consonant on each edge of the word. Using our notation, this pattern can be represented as follows:

(18) (C|@) ((C|@) (C|@)V(C|@) (C|@)) ((C|@) (C|@)V(C|@) (C|@))*(C|@)

The ampersand indicates ∈; thus each instance of (C—@) indicates an optional consonant. The pattern also includes an extra set of parentheses around the first syllable. These are for convenience and have no effect on the FSA constructed.

There are two ways to create this. One is to create the FSA directly with a single application of re.

(19) re -fedmxr…>syl.xml

We replace …with the quoted pattern in (18). The flags here indicate that we read in a regular expression (-r), make an FSA out of it (-f), prune ∈-arcs (-e), determinize (-d), minimize (-m), and save (-x).

We can also construct the FSA in steps using other programs from the suite. The full sequence is given in (20):

(20) a. `re -fdemxr '(C|@)' > c.xml`

b. `re -fdemxr '(C|@)(C|@)V(C|@)(C|@)' > syl1.xml`

c. `kleene syl1.xml > sylrep.xml`

d. `concatenate c.xml syl1.xml > w1.xml`

e. `concatenate w1.xml sylrep.xml > w2.xml`

f. `concatenate w2.xml c.xml > syl.xml`

In (20a), we use re to create an FSA that represents an optional consonant. In (20b), we use re again to create the FSA for a single syllable. In (20c), we create an FSA for any number of syllables. In steps (20d,e,f), we concatenate all the pieces together: an optional consonant, a single syllable, any number of syllables, and another optional consonant.

We can now test our FSA with re -i. Some tests with acceptable strings are given in (20).

(21) `re -i syl.xml -s CVCC`

`re -i syl.xml -s CVCCVCCC`

`re -i syl.xml -s CVCCCCVCC`

Some tests with illegal strings are given in (22):

(22) `re -i syl.xml -s CCC`

`re -i syl.xml -s CVCCVCCCC`

`re -i syl.xml -s CVCCCCCVCC`

We now show how to obtain the morphological FSA in (3).[6.] The pattern here is a little trickier. The basic generalization is that a root like *nation* can be followed by up to three suffixes: *-al*, *-ize*, and *-ation*. Each one requires the presence of the preceding one and the pattern is unbounded. We have this regular expression:

(23) `nation(alization)*(al(ize|@)|@)`

The root comes first. This is followed by any number of repetitions of the entire sequence, terminated by a partial sequence of one or two affixes. The ampersands indicate optionality and the nesting of parentheses captures the fact that *-ize* requires the presence of *-al*.

Notice too that this expression finesses a subtle orthographic issue. The suffix *-ize* truncates its final letter when followed by *-ation*. This restriction is easily captured in the pattern above. It also means that there would not be much to gain if we were to try to assemble the pattern incrementally as we did in (20) is the code to build the FSA in one fell swoop with re.

(24) re -fdemxr …>morph.xml

Once again, we replace …with the quoted pattern in (23). The flags are the same as in the previous example.

As with the previous example, we can test our FSA with re -s. The examples in (25) test legal strings.

```
(25)  re -i morph.xml -s nation
      re -i morph.xml -s nationalize
      re -i morph.xml -s nationalizationalizational
```

The examples in (26) test illegal strings.

```
(26)  re -i morph.xml -s nationize
      re -i morph.xml -s nationalation
      re -i morph.xml -s nationationalizational
```

Much more complex FSAs can be built directly, from lists of strings, or incrementally. Here we have only provided a brief exemplification of what can be done.

V. OTHER TOOLKITS

There are a number of other finite-state toolkits available. The FSP toolkit is not intended to compete with any of these, but to provide different functionality in a different package so that users with different needs have the fullest range of options. In this section, we review some of the more well-known toolkits.

Xerox Finite State Toolkit A robust and very efficient package. Provides support for transducers as well. The downside is that this is a commercial product. In addition, it is not always clear what algorithms are implemented (http://www.xrce.xerox.com).

AT&T FSM library Another very robust and efficient package. Provides support for transducers and for weighted automata. The downside is that it's a commercial product and not the easiest software to use (http://www.research.att.com/~fsmtools/fsm/).

Finite State Automata utilities An open-source free toolkit. Does not appear to include support for transducers or for weights. The full package requires Prolog and Tcl/tk (http://odur.let.rug.nl/~vannoord/Fsa/).

JFLAP A free java-based teaching package for formal language theory that goes well-beyond FSAs. Very good for stepwise construction of FSAs and for visualizing automata. The notation used is idiosyncratic and export of results or images is tricky (http://www.jflap.org/).

VI. PROGRAMMING CHALLENGES

The FSP package is far from complete and there were a number of programming challenges in creating the package and a number of things to do to make it more complete.

Command-line vs. GUI The programs are currently written for a UNIX-style command-line. This works fine for Unix systems or for Mac OS, but does not accommodate Windows users.[7.] A nice next step would be the addition of a graphical user interface.

Scalability The programs are written in C for efficiency and speed, and the algorithms are correctly implemented. However, the programming is, in some cases, quite inefficient. This results in a dramatic drop in performance/speed with large FSAs. Another nice next step would be to use more efficient data structures so as to address this.

Programming tools The programs are written in C, but also make use of the lex/flex and bison/yacc grammar tools. In addition, the suite uses the expat XML parser for reading in XML representations of FSAs and standard GNU libraries for input/output. These tools are not available on all systems and greater portability could be obtained by eliminating these dependencies.

Compiling The programs are distributed as source C code with pre-built binaries for Intel Macs. It would be ideal to distribute binaries for other architectures, especially Windows.

Weights As indicated above, the system of weights is only partially implemented.

VII. CONCLUSION

In this paper, we have described the Finite-State Playground suite of programs. We've explained their basic functionality and we've provided several examples of how we can construct linguistically-relevant automata and use those automata to validate natural language expressions. We've also discussed some of the other finite-state modeling packages available and some of the programming challenges yet to be faced.

NOTES

[1.] www.graphviz.org
[2.] This is highly simplified. A full treatment of English syllabification can be found in Hammond (1999).
[3.] All FSAs depicted in this paper have been rendered in this way.
[4.] All of the FSA images in this paper were produced using this software.
[5.] One exception is that the re program allows for certain flags to be combined. For example, one can create an FSA, remove □-arcs, determinize, and minimize in one application with the flag combination -fedm.
[6.] Note that the automaton as depicted is schematized. Each individual letter has its own arc in the actual FSA. Also, the facts of English affixation are simplified greatly for this example.
[7.] It may be that the package will compile for Windows with a Unix emulation layer like Cygwin or MinGW.

REFERENCES

Bird, S. (1995). *Computational Phonology*. Cambridge: Cambridge University Press.

Bird, S. & Ellison, T. M. (1994). One-level phonology: autosegmental representations and rules as finite automata. *Computational Linguistics*, 20, 55–90.

Brzozowski, J. (1962). Canonical regular expressions and minimal state graphs for definite events. *Mathematical theory of Automata*, 12, 529–561.

Ellison, T. M. (1994). Phonological derivation in Optimality Theory. *COLING 94*, 1007–1013.

Hammond, M. (1999). *The Phonology of English*. Oxford: Oxford University Press.

Hopcroft, J. & Ullman, J. (1979). *Introduction to Automata Theory, Languages, and Computation*. Reading: Addison-Wesley.

Johnson, D. C. (1972). *Formal aspects of phonological description*. The Hague: Mouton.

Kaplan, R. & Kay, M. (1994). Regular model of phonological rule systems. *Computational Linguistics*, 20, 331–378.

Karttunen, L. (1983). KIMMO: a general morphological processor. *Texas Linguistic Forum*, 22, 163–186.

Karttunen, L. (1998). The proper treatment of optimality in computational phonology. In *The Proceedings of FSMNLP '98: International Workshop on Finite-State Methods in Natural Language Processing*, Ankara, Turkey: Bilkent University, pp. 1–12.

Kleene, S. C. (1956). Representation of events in nerve nets and finite automata, automata studies. *Ann. Math. Studies*, 34, 3–41.

Lawson, M. V. (2004). *Finite Automata*. Boca Raton: Chapman & Hall/CRC.

McCulloch, W. & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5, 115–133.

Turing, A. (1936). On computable numbers: with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 230–265.

APPENDIX: COMPILING THE SOFTWARE

The software is distributed over the web from `http://www.u.arizona.edu/~hammond/`. The package includes instructions for installing the software and documentation of all the programs in HTML and man-file formats.

To install the software, follow the following steps:

1. Download the package.
2. Uncompress the package: `gunzip flbi1.02.tar.gz`.
3. Extract the package from the tar archive: `tar xf flbi1.02.tar`.
4. If your system is Intel Mac, you are done. The programs are all located in the `flbi/bin` subdirectory.
5. If not, switch to the `flbi/src` directory and type `make`. If you do not have all the required packages or they are located in other places, then you may need to tweak the `Makefile` accordingly. The `src` subdirectory also includes a file `linux.makefile` which includes different defaults and which might be more useful than the default `Makefile`, depending on your system configuration.

