# XFST2FSA: Comparing Two Finite-State Toolboxes

Yael Cohen-Sygal and Shuly Wintner
Department of Computer Science
University of Haifa

July 30, 2005

# Introduction

- Motivation: finite state techniques and toolboxes
- The XFST2FSA compiler:
  - Compilation process
  - Problems and solutions
- Comparison of XFST and FSA:
  - Usability
  - Performance
- Conclusions and future work

# Motivation

Finite-state technology is widely considered to be the appropriate means for describing the phonological and morphological phenomena of natural languages

- Descriptive power
- Closure properties $\Rightarrow$ modularity
- Computational efficiency

# Motivation

Finite-state toolboxes:

- Provide a language for extended regular expressions
- Include a compiler from regular expressions to finite state devices, automata and transducers
- Include efficient implementations of algorithms for closure properties, minimization, determinization, etc.
- Implement special operators that are useful for linguistic description.

Unfortunately, there are no standards for the syntax of extended regular expression languages and switching from one toolbox to another is a non-trivial task.

# XFST vs. FSA Utils

| | XFST | FSA Utils |
|---|---|---|
| standard operators | + | + |
| advanced operators | replacement<br>markup<br>restriction | – |
| advanced methods | compile-replace<br>Flag diacritics | weighted networks<br>Prolog predicates |
| visualization | – | + |
| availability | proprietary | free, open source |

# The XFST2FSA compiler

- Motivation: finite state techniques and toolboxes
- The XFST2FSA compiler:
  - Compilation process
  - problems and solutions
- Comparison of XFST and FSA:
  - Usability
  - Performance
- Conclusions and future work

# The XFST2FSA compiler

- *XFST2FSA*: a compiler which translates XFST grammars into grammars in the language of FSA Utils
- Strong parallelism between the languages
- Certain constructs are harder to translate and require more innovation, e.g., replacement, markup and restriction
- We focus on the core of the finite state calculus: naïve automata and transducers (no weights and advanced methods).

# Examples

```
! XFST grammar for describing English noun pluralization
! English vowels
define vowel a|e|i|o|u;

! Nouns lexicon
define noun {book}|{case}|{box}|{watch}|{glass}|{copy}|{guy};
! Suffix with s
define AddS noun []:[%+ s];
! If the noun ends with x, ch or s
define esException %+ -> e || x | [c h] |s _;
! If the noun ends with y precedded by non-vowel symbol
define yException [y %+] -> [i e] || \vowel _;
! Basic pluralization
define normal %+ -> [];
! The complete network
define plural AddS .o. esException .o. yException .o. normal;
regex plural;
```

# Examples

```
%% This file contains the fsa code for the xfst code in exa1.xfst.
:- multifile macro/2.
:- multifile rx/2.
%%  Load macros in macros.pl
:- ensure_loaded(macros).

:- user:bb_put(fsa_regex_cache:vowel,on).
:- user:bb_put(fsa_regex_cache:noun,on).
:- user:bb_put(fsa_regex_cache:AddS,on).
:- user:bb_put(fsa_regex_cache:esException,on).
:- user:bb_put(fsa_regex_cache:yException,on).
:- user:bb_put(fsa_regex_cache:normal,on).
:- user:bb_put(fsa_regex_cache:plural,on).
%%  XFST grammar for describing English noun pluralization
%%  English vowels
macro(vowel,{'a' , 'e' , 'i' , 'o', 'u'}).
```

# Examples

```
%%  Nouns lexicon
macro(noun,{['b','o','o','k',[]], ['c','a','s','e',[]],
   ['b','o','x',[]], ['w','a','t','c','h',[]],
   ['g','l','a','s','s',[]], ['c','o','p','y',[]], ['g','u','y',[]]}).
%%  Suffix with s
macro(AddS,['noun',([]):((['+','s']))]).
%%  If the noun ends with x, ch or s
macro(esException,cond_rep_or_or(('+'),('e'),
            ({'x' , (['c','h']),'s'}),([]))).
%%  If the noun ends with y precedded by non-vowel symbol
macro(yException,cond_rep_or_or(((['y','+'])),((['i','e'])),
            (~ ('vowel') & ?),([]))).
%%  Basic pluralization
macro(normal,uncond_rep(('+'),([]))).
%%  The complete network
macro(plural,((('AddS') o ('esException')) o
               ('yException')) o ('normal')).
macro(regex,'plural').
```

# Examples

```
! XFST grammar for Arabic nominative definite and indefinite nouns
! The lexicon - Arabic nouns
define noun {qammar} | {kitaab} | {%$ams} | {daftar};

! Indefinite nouns: add un suffix
define indefinite noun []:[u n];

! definite nouns: add ‘al prefix and u suffix
define definite []:[%‘ a l] noun []:[u];

! Assimilation: the ‘l’ in the prefix assimilates with the first
! letter of the noun when the consonant is $, d, etc.
define shAssim l -> %$ || .#. %‘ a _ %$;
define dAssim l -> d  || .#. %‘ a _ d ;

define Arabic [definite .o. shAssim .o. dAssim] | [indefinite];
regex Arabic;
```

# Examples

```
%% This file contains the fsa code for the xfst code in exa3.xfst.

:- multifile macro/2.
:- multifile rx/2.
%%  Load macros in macros.pl
:- ensure_loaded(macros).

:- user:bb_put(fsa_regex_cache:noun,on).
:- user:bb_put(fsa_regex_cache:indefinite,on).
:- user:bb_put(fsa_regex_cache:definite,on).
:- user:bb_put(fsa_regex_cache:shAssimilation,on).
:- user:bb_put(fsa_regex_cache:dAssimilation,on).
:- user:bb_put(fsa_regex_cache:ArabicExample,on).
```

# Examples

```
%%  XFST grammar for Arabic nominative definite and indefinite nouns
%%  The lexicon - Arabic nouns
macro(noun,{['q','a','m','m','a','r',[]] ,
            ['k','i','t','a','a','b',[]] ,
            ['$','a','m','s',[]],['d','a','f','t','a','r',[]]}).
%%  Indefinite nouns: add un suffix
macro(indefinite,['noun',([]):((['u','n']))]).
%%  definite nouns: add `al prefix and u suffix
macro(definite,[([]):((['`' , 'a','l'])) , 'noun',([]):(('u'))]).
%%  Assimilation: the `l' in the prefix `al assimilate with the first
%%  letter of the noun when the consonant is $, d, etc.
macro(shAssim,cond_rep_or_or_start(('l'),('$'),(['`','a']),('$'))).
macro(dAssimi,cond_rep_or_or_start(('l'),('d'),(['`','a']),('d'))).
macro(ArabicExample,{((('definite') o ('shAssim')) o
                       ('dAssim')),('indefinite')}).
macro(regex,'ArabicExample').
```

# Compilation process

1. The XFST grammar is parsed, and a tree representing its syntax is created
   - A specification of XFST syntax is needed...
   - but is unavailable
2. Traversing the tree, the equivalent FSA grammar is generated
   - A specification of XFST semantics is needed...
   - but is not fully available

# Compilation: basic operators

| XFST syntax | FSA syntax | Meaning |
|---|---|---|
| A* | A* | Kleene star |
| A \| B | {A,B} | union |
| A & B | A & B | intersection |
| A - B | A - B | A minus B |
| A/B | ignore(A,B) | A ignoring B |
| $A | $A | containment |
| A B | [A,B] | concatenation |
| A^n | does not exist | n-ary concatenation |
| A.x.B | A x B | crossproduct |
| A.o.B | A o B | composition |
| (A) | A^ | optionality |
| [ ] | ( ) | precedence |
| R.i | invert(R) | regular relation inverse |

# Compilation: advanced operators

- Include replacement, markup and restriction
- Have no equivalents in FSA, and therefore have to be implemented from scratch
- This was done using existing documentation.

# Compilation: advanced operators

Problem 1: not all operators are fully documented

- The operator `A@<-B` (obligatory, lower to upper, left to right, longest match replacement) is not documented. However:
- The operator `A<-B` (obligatory, lower to upper replacement ) is defined as `[B->A].i` (where `B->A` is the obligatory, upper to lower replacement of the language `B` by the language `A`).
- Conclusion: `A@<-B` is constructed as `[B@->A].i` (where `[B@->A]` is the obligatory, upper to lower, left to right, longest match replacement of the language `B` by the language `A`).
- The construction of the operator `B@->A` is documented.

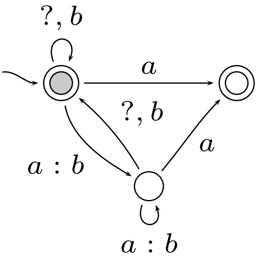
# Compilation: advanced operators

Problem 2: for some of the documented operators, the published algorithms are erroneous in some special cases

- Consider the replace operator `A->B || L _ R` (conditional replacement of the language `A` by the language `B`, in the context of `L` on the left and `R` on the right side, where both contexts are on the upper side).
- Consider a rule of the form `A->B || _ ?`, where `A` and `B` are some regular expressions denoting languages.
- This rule states that any member of the language `A` on the upper side is replaced by all members of the language `B` on the lower side when the upper side member is not followed by the end of the string on which the rule operates.

# Compilation: advanced operators

- For example, the rule `a->b || _ ?` is expected to generate the following automaton:

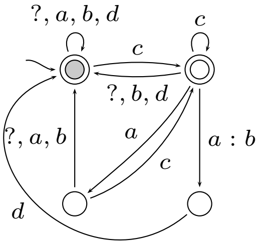



- However, a direct implementation of the documented algorithms always yields a network accepting the empty language, independently of the way `A` and `B` are defined.

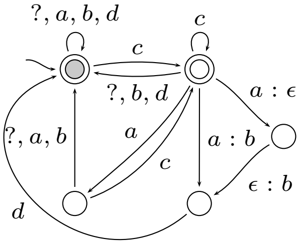
# Compilation: advanced operators

Problem 3: in some cases XFST produces networks that are somewhat different from the ones in the literature: the relations (as sets) are equal but the resulting networks (as graphs) are not isomorphic.
For example, consider the replace rule `a->b || c _ d`

# Compilation: advanced operators




XFST network




Self-implemented network
(by the documented algorithms)

# Compilation: advanced operators

- In some cases multiple accepting paths are obtained
- This is probably a result of adding $\epsilon$-self-loops in order to deal correctly with $\epsilon$-symbols in composition
- The multiple paths can then be removed using filters
- Presumably, XFST implements this strategy
- This solution requires direct access to the underlying network, and cannot be applied at the level of the regular expression language.

# Validation of correctness

- Ideally: check that the obtained FSA networks are equivalent to the XFST ones from which they were generated
- Unfortunately, this is only possible for very small networks
- Therefore, validation strategy:
  - Check each operator independently for several instances
  - Test the compiler on a large-scale grammar: HAMSAH
  - Exhaustive tests produced the same outputs for both networks.

# Comparison

- Motivation: finite state techniques and toolboxes
- The XFST2FSA compiler:
  - Compilation process
  - problems and solutions
- Comparison of XFST and FSA:
  - Usability
  - Performance
- Conclusions and future work

# Comparison of XFST and FSA: Usability

| | XFST | FSA |
|---|---|---|
| display formats | text (limited) | text<br>GUI |
| save as | binary | binary, text, PostScript |
| Code generation | – | C, C++, Java, Prolog |

# Comparison of XFST and FSA: performance

- A true comparison of the two systems should compare two different grammars, each designed specifically for one of the two toolboxes, yielding the same comprehensive network
- However, as such grammars are not available, we compare the two toolboxes using a grammar designed and implemented in XFST and compiled into FSA.

# Comparison of XFST and FSA: performance

HAMSAH:

- Approximately 2 million states and 2.2 million arcs
- Hebrew adjectives: approximately 100,000 states and 120,000 arcs
- Hebrew nouns: approximately 700,000 states and 950,000 arcs.
- Each network created by composing a series of rules over a large-scale lexicon
- Significant usage of replace rules and compositions
- Grammars compiled and executed on a 64-bit computer with 16Gb of memory.

# Comparison of XFST and FSA: performance

| | | FSA | | XFST | |
|---|---|---|---|---|---|
| | | Time | Space | Time | Space |
| Compilation | Full | 13h 43m | 11Gb | 27m 41s | 3Gb |
| | nouns | 2h 29m | | 11m 4s | |
| | adjectives | 14m 56s | | 8m 21s | |
| Analysis | Full, 350 words | – | | 5s | |
| | nouns, 120 | 1h 50m | | 0.17s | |
| | adjectives, 50 | 2m 34s | | 0.17s | |

# Conclusion

- Motivation: finite state techniques and toolboxes
- The XFST2FSA compiler:
  - Compilation process
  - problems and solutions
- Comparison of XFST and FSA:
  - Usability
  - Performance
- Conclusions and future work

# Conclusion

Contributions:

- Facilitating the use of grammars developed with XFST on publicly available systems
- Providing a closer insight into the theoretical algorithms which XFST is based on
- A full implementation, in FSA, of most of XFST's operators
- Investigation of two similar, but different systems, facilitating a comparison on compatible benchmarks.

# Future work

- Construct more XFST operators in FSA
- Locate more boundary cases in replace rules
- Convert XFST grammars into other formalisms (FSM)
- FSA2XFST...