Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Research Report
RR-02-04

DFKI Finite-State Machine Toolkit

Jakub Piskorski

July 2002

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 2080
67608 Kaiserslautern, FRG
Tel: +49 (631) 205-3211
Fax: +49 (631) 205-3210
E-Mail: info@dfki.uni-kl.de

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel: +49 (631) 302-5252
Fax: +49 (631) 302-5341
E-Mail: info@dfki.de

WWW: http://www.dfki.de

Deutsches Forschungszentrum für Künstliche Intelligenz

DFKI GmbH

German Research Centre for Artificial Intelligence

Founded in 1988, DFKI today is one of the largest non-profit contract research institutes in the field of innovative software technology based on Artificial Intelligence (AI) methods. DFKI is focusing on the complete cycle of innovation — from world-class basic research and technology development through leading-edge demonstrators and prototypes to product functions and commercialisation.

Based in Kaiserslautern and Saarbrücken, the German Research Centre for Artificial Intelligence ranks among the important "Centres of Excellence'' world-wide.

An important element of DFKI's mission is to move innovations as quickly as possible from the lab into the marketplace. Only by maintaining research projects at the forefront of science can DFKI have the strength to meet its technology transfer goals.

DFKI has about 165 full-time employees, including 141 research scientists with advanced degrees. There are also around 95 part-time research assistants.

Revenues for DFKI were about 30 million DM in 2000, from government contract work and from commercial clients. The annual increase in contracts from commercial clients was greater than 20% during the last three years.

At DFKI, all work is organised in the form of clearly focused research or development projects with planned deliverables, various milestones, and a duration from several months up to three years.

DFKI benefits from interaction with the faculty of the Universities of Saarbrücken and Kaiserslautern and in turn provides opportunities for research and Ph.D. thesis supervision to students from these universities, which have an outstanding reputation in Computer Science.

The key directors of DFKI are Prof. Wolfgang Wahlster (CEO) and Dr. Walter Olthoff (CFO).

DFKI's five research departments are directed by internationally recognised research scientists:

- ❏ Knowledge Management (Director: Prof. A. Dengel)
- ❏ Intelligent Visualisation and Simulation Systems (Director: Prof. H. Hagen)
- ❏ Deduction and Multiagent Systems (Director: Prof. J. Siekmann)
- ❏ Language Technology (Director: Prof. H. Uszkoreit)
- ❏ Intelligent User Interfaces (Director: Prof. W. Wahlster)

In this series, DFKI publishes research reports, technical memos, documents (e.g. workshop proceedings), and final project reports. The aim is to make new results, ideas, and software available as quickly as possible.

Prof. Wolfgang Wahlster

Director

DFKI Finite-State Machine Toolkit

Jakub Piskorski

DFKI-RR-02-04

The work presented in this report has been supported by a research grant from the German Federal Ministry of Education, Science, Research and Technology (Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie) to the DFKI projects: PARADIME, FKZ ITW 9704, and WHITEBOARD, FKZ: 01 IW 002.

ISSN 0946-008X

DFKI Finite-State Machine Toolkit

Jakub Piskorski

Language Technology Department
German Research Center for Artificial Intelligence
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
phone: +49 681 302-5306
email: piskorsk@dfki.de

ABSTRACT: Finite-state devices such as finite-state automata and finite-state transducers have been known since the emergence of computer science and are recently extensively used in many areas of language technology. The use of finite-state devices is mainly motivated by their time and space efficiency. In this paper we present the Finite-State Machine Toolkit for building, combining and optimizing the finite-state machines, developed at the Language Technology Lab of the German Research Center for Artificial Intelligence.

1. Overview

Finite-State devices such as finite-state automata and finite-state transducers have been known since the emergence of computer science and have been successfully applied in many areas such as compiler construction, pattern matching, cryptography, data compression and switching theory. Recently, finite-state technology has been widely applied to various domains of natural language processing, including phonology [Kaplan and Kay, 94], construction of lexical and morphological analyzers [Koskenniemi, 83], [Silberztein, 93], part-of-speech filtering [Roche and Schabes, 95], shallow parsing [Abney, 96], [Pavia, 99], [Greffenstette, 96], [Ait-Mohtar and Chanod, 97] and speech recognition [Pereira and Riley, 97], [Mohri, 97]. The use of finite-state devices is mainly motivated by their time and space efficiency [Lewis and Papadimitriou, 81]. From the linguistic point of view, most of the local language phenomena can be easily and intuitively expressed as finite-state devices [Roche and Schabes, 97].

Over the last few years, there has been ongoing research work centered around application of finite-state technology in NLP at the Language Technology Lab at the German Research Center for Artificial Intelligence (DFKI). This includes amongst others finite-state based shallow text processing systems, SMES [Neumann, et. al., 97] based on an ATN compiler [Krieger, 95], and SPPC – shallow text processing system for German [Piskorski and Neumann, 00], morphological parsing of Japanese [Siegel and Scherf, 00], regular approximation of context-free grammars [Mohri and Nederhof, 01], [Nederhof, 01], and core finite-state tools for manipulating finite-state devices [Scherf, 00].

In this paper we present the Finite-State Machine Toolkit (FSM Toolkit) – a library of tools for constructing, combining and optimizing finite-state machines, developed at the

Language Technology Lab of DKFI. Finite-state machines are generalizations of better known weighted finite-state automata and weighted finite-state transducers. The architecture and functionality of the DFKI FSM Toolkit is mainly based on the tools developed by AT&T [Mohri et. al., 00], and it provides all state-of-the-art transformation and optimization operations for finite-state devices. Additionally, the toolkit includes implementations of some recently presented operations (e.g., incremental construction of minimal deterministic finite-state automata [Daciuk, 98]) and modifications of some of the state-of-the-art algorithms (e.g., removal of epsilon transitions, weighted version of the local extension algorithm). The Toolkit runs on both Unix and MS Windows platforms and constitutes the backbone of the earlier mentioned system SPPC and a named-entity recognition platform presented in [Piskorski et. al., 02B].

The rest of this paper is organized as follows. In section 2 we introduce the basic definitions of finite-state devices including in particular finite-state machine which is an underlying model for the FSM Toolkit. Actually, there are many definitions of automata and transducers, but the differences are mainly notational and therefore not important from a scientific point of view. The definitions presented here are based on the definitions in [Hopcroft and Ullman, 79], [Mohri, 97] and [Roche and Schabes, 96], which are regarded as standard references for finite-state devices. In section 3 we present the FSM Toolkit, including detailed description of provided operations and comparison with other similar finite-state packages. Finally, a quick user guide to the FSM Toolkit is presented in Appendix A.

2. Finite-State Devices

2.1 Finite-State Automata

A ***finite-state automaton*** is a device that can be in one of a finite number of ***states***. Under certain conditions it switches from one state into another. The set of states contains an ***initial state***, ***final states*** and other states. The automaton starts working in its initial state and processes a sequence of ***input symbols***. The symbols must come from a finite set of symbols, usually called an ***alphabet*** of the automaton. The interpretation of the symbols depends on the task which has to be solved. Processing of the input symbols consists of a sequence of moves, where in each move the automaton consumes (reads) the next symbol and possibly switches to another state, depending on the current state and current symbol. The set of so called ***transitions*** of the automaton defines for each state how the automaton may be switched from this state into another one. A single transition consists of source state, target state, and input symbol, where the latter is usually called the input label of the transition. In a given state, the automaton tests whether the next input symbol and current state match any of the transitions for this state. If they do, the automaton switches to the target state of the transition. Obviously, there may be more than just one valid transition from a given state. In such case, the transition is chosen arbitrarily. In case there are no transitions that match the current input and the current state, the automaton stops and ***rejects*** the input. Further, it is also possible to move from one state to another without consuming any input symbol. Such non-consuming transitions are labeled with ε (***epsilon transitions***). If the automaton succeeds in consuming

the input completely and is in a final state, then we say that it ***accepts*** the input. The set of all sequences of symbols which are accepted by a given automaton is called the ***language accepted*** by this automaton.

Let us now formally define a finite-state automaton. A finite set of symbols is denoted as an alphabet Σ. As mentioned above, an ***empty string*** is represented as ε. Let Σ be an alphabet and $v = v_1 v_2 ... v_n$, $u = u_1 u_2 ... u_m$ two strings, where $v_1, v_2, ..., v_n, u_1, u_2, ..., u_m \in \Sigma$. Then we define the ***concatenation*** of two strings as:

$$v \circ u = v_1 v_2 ... v_n u_1 u_2 ... u_m \tag{1}$$

Concatenation can be easily extended to sets. Further, we define the set of all possible words over the alphabet Σ as:

$$\Sigma^* = \{v \mid v = v_1 v_2 ... v_n \text{ for } n \geq 1 \text{ and } v_i \in \Sigma \text{ for all } 1 \leq i \leq n\} \cup \{\varepsilon\} \tag{2}$$

For $w \in \Sigma^*$ with $w = w_1 w_2 ... w_n$ we define its ***reversion*** as w^R:

$$w^R = w_n w_{n-1} ... w_1 \tag{3}$$

Definition 1 A ***finite-state automaton*** (FSA) is a 5-tuple $M = (Q, \Sigma, \delta, i, F)$, where Q is a finite set of states, $i \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, Σ is a finite alphabet and $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ is the transition function.

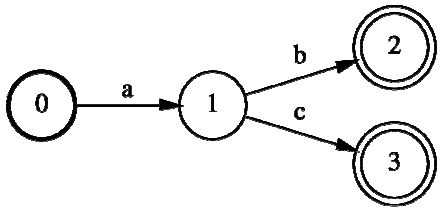
Further, we extend the transition function δ to $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ to accept words over Σ as argument. We denote the size of the automaton M with $|M|$, which is equal to the number of states $|Q|$. We denote the language accepted by the automaton M as $L(M)$ and define it as:

$$L(M) = \{v \in \Sigma^* \mid \hat{\delta}(i, v) \cap F \neq \varnothing\} \tag{4}$$

Automata can be represented as directed graphs, where the nodes denote states and edges represent transitions. We use positive natural numbers for labeling the nodes. In order to extend the standard convention for visualizing graphs, we represent the final states as two concentric circles and the initial state is represented by a circle drawn with a thicker line. Note that an initial state may be simultaneously a final state. Example 1 presents a simple automaton and its corresponding visual representation.

Example 1 Let $M = (Q_M, \Sigma_M, \delta_M, i_M, F_M)$ be an FSA, where $Q_M = \{0,1,2,3\}$, $\Sigma_M = \{a, b, c\}$, $\delta_M = \{(0, a, 1), (1, b, 2), (1, c, 3)\}$, $i_M = 0$ and $F_M = \{2,3\}$.

The automaton M accepts the language $L(M) = \{ab, ac\}$.



We say that an automaton is $\varepsilon-$***free*** if it has no transitions labeled with ε. An automaton is said to be ***deterministic*** if it is $\varepsilon-$*free* and for each state there is at most one transition labeled with the same symbol, otherwise it is said to be ***nondeterministic***. Every nondeterministic automaton may be converted to an equivalent deterministic automaton, which accepts the same language [Hopcroft and Ullman, 79]. Among many automata that accept the same language there exists one which has a minimal number of states (has minimal size) [Hopcroft and Ullman, 79]. It is called a ***minimal automaton***. The automaton presented in example 1 is deterministic. Further, it is the minimal automaton that accepts the language $\{ab,ac\}$. Finally, the FSAs are closed under union, Kleene-star, concatenation, intersection and complementation, thus allowing for natural and flexible descriptions [Roche and Schabes, 96].

2.2 Finite-State Transducers

Finite-state transducers (FST) are automata for which each transition has an ***output label*** in addition to the more familiar input label. Transducers transform (transduce) input strings into output strings. The output symbols come from a finite set, usually called ***output alphabet***. Since the input and output alphabet are frequently the same, there is usually no distinction between them. A formal definition of a transducer follows in definition 2.

Definition 2 A ***finite-state transducer*** (FST) is a 5-tuple $M=(Q,\Sigma,E,i,F)$, where Q is a finite set of states, $i\in Q$ is the initial state, $F\subseteq Q$ is a set of final states, Σ is a finite alphabet and $E:Q\times(\Sigma\cup\{\varepsilon\})\times\Sigma^*\times Q$ is the set of transitions (arcs).

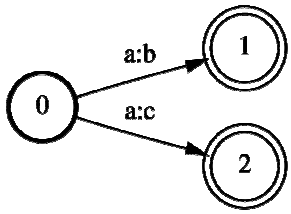
Further, we define the ***state transition function*** $\delta:Q\times(\Sigma\cup\{\varepsilon\})\to 2^Q$ as follows:

$$\delta(p,a)=\{q\in Q\mid\exists v\in\Sigma^*:(p,a,v,q)\in E\} \tag{5}$$

The ***emission function*** $\lambda:Q\times(\Sigma\cup\{\varepsilon\})\times Q\to 2^{\Sigma^*}$ is defined as:

$$\lambda(p,a,q)=\{v\in\Sigma^*\mid(p,a,v,q)\in E\} \tag{6}$$

Example 2 Let $M=(Q_M,\Sigma_M,E_M,i_M,F_M)$ be an FST, where $Q_M=\{0,1,2\}$, $\Sigma_M=\{a,b,c\}$, $\delta_M=\{(0,a,b,1),(0,a,c,2)\}$, $i_M=0$ and $F_M=\{1,2\}$



M transduces *a* to *b* or *a* to *c*. Note that for visualizing transducers we use the colon to separate the input and output labels of a transduction.

The emission function λ may be extended to $\hat{\lambda}: Q \times (\Sigma \cup \{\varepsilon\})^* \times Q \to 2^{\Sigma^*}$. It takes as an argument strings over Σ and emits a set of all possible output strings. A transducer $M = (Q, \Sigma, E, i, F)$ can be seen as a mapping $f: \Sigma^* \to 2^{\Sigma^*}$, where $u \in f(v)$ if and only if $\exists q \in F$ with $u \in \hat{\lambda}(i, v, q)$. We denote such mapping for a transducer *M* as $\vec{M}$. For the transducer in example 2, $\vec{M}(a) = \{b, c\}$.

Definition 3 A mapping $f: \Sigma^* \to 2^{\Sigma^*}$ is called ***rational transduction*** if there exists a FST *M* such that $f = \vec{M}$. If $\forall v \in \Sigma^*$, $|\vec{M}(v)| \leq 1$ then $\vec{M}$ is called ***rational function***.

Since transducers compute functions, a slightly different definition of being deterministic is used. A FST *M* is said to be ***deterministic*** if $\forall q, p \in Q, \forall v \in \Sigma$: $|\delta(p, v)| \leq 1$, $|\lambda(p, v, q)| \leq 1$, $|\delta(p, \varepsilon)| = 0$, and $|\lambda(p, \varepsilon, q)| = 0$. The transducer in the example 2 is not deterministic. The term ***subsequential transducer*** is often used as a synonym for deterministic transducer. However, the formal definition of subsequential transducers is slightly different [Mohri, 97]. Subsequential transducers are seven tuples $M = (Q, \Sigma, i, F, \delta, \lambda, \rho)$, where Q, Σ, i, F are defined as before and $\delta: Q \times \Sigma \to Q$ is the ***deterministic transition function***, $\lambda: Q \times \Sigma \to \Sigma^*$ is the ***deterministic emission function***, and $\rho: F \to \Sigma^*$ is the ***final emission function***. The final emission function is used to output strings when the automaton has consumed the input string and is in a final state. In [Mohri, 97] ***p-subsequential transducers*** were introduced, which allow emission of up to *p* final output strings. They proved to be very useful for describing linguistic ambiguities.

Contrary to finite-state automata, finite-state transducers have weaker closure properties, e.g., they are not closed under intersection. Further, only a certain subclass of transducers are determinizable [Mohri, 97]. ***Weighted finite-state automata*** (WFSA) or ***weighted finite-state transducers*** (WFST) are automata or transducers in which each transition has a weight as well as input/output labels. Instead of defining these devices formally, in the next section the more general device, the finite-state machine, will be introduced.

2.3 Finite-State Machines

A ***finite-state machine*** (FSM) is a generalization of FSAs, WFSAa, FSTs and WFSTs. It is used as an underlying model for the finite-state tools presented in section 3. In contrary to simple transducers (FSTs), an FSM distinguishes between input and output alphabet and

allows only single symbols as output labels (***letter transducer***). Additionally, a weight is assigned to each transition in an FSM. The ***semirings*** are used as a basis in the theory of weighted finite-state devices [Kuich and Salomaa, 86]. In FSMs, the choice of a semiring determines the interpretation (computation) of weights.

Definition 4 A ***closed semiring*** is a system $(S, \oplus, \otimes, \bar{0}, \bar{1})$, where S is a possibly infinite set of elements, $\oplus$ (the ***summary operator***) and $\otimes$ (the ***extension operator***) are binary operations on S, and $\bar{0}$ and $\bar{1}$ are elements of S, satisfying the following properties:

1. $(S, \oplus, \bar{0})$ and $(S, \otimes, \bar{1})$ are monoids, which means that S is closed under $\oplus$ and $\otimes$, both operations are associative, i.e., $\forall a,b,c \in S: (a \oplus b) \oplus c = a \oplus (b \oplus c)$ and $(a \otimes b) \otimes c = a \otimes (b \otimes c)$, and $\bar{0}$ and $\bar{1}$ are identities of $\oplus$ and $\otimes$.
2. $\oplus$ is commutative: $\forall a,b \in S: a \oplus b = b \oplus a$
3. $\bar{0}$ is an annihilator: $\forall a \in S: a \otimes \bar{0} = \bar{0} \otimes a$
4. $\oplus$ is idempotent: $\forall a \in S: a \oplus a = a$
5. $\otimes$ distributes over $\oplus$: $\forall a,b,c \in S:$ $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$

If the fourth property is not satisfied, we call such a system a ***semiring***. The systems $(R_+, +, \cdot, 0, 1)$ and $(R \cup \infty, \min, +, \infty, 0)$[1] are examples of semirings, where the latter one is called a ***tropical semiring***. Note that the tropical semiring is closed in contrary to $(R_+, +, \cdot, 0, 1)$, since addition is not idempotent (whereas the min operation is).

An FSM follows a path corresponding to a given input string and emits an output string plus a weight obtained by combining the weights along this path. The extension operator $\otimes$ of the semiring is used for combining the weights on a single path. In case the FSM is not deterministic, there might exist more than one corresponding path for a given input string. Furthermore, different paths might produce different output strings. In such a situation, the summary operator $\oplus$ is used to combine the weights of paths yielding the same output string for a given input string. Thus, an output of an FSM for a given input string can be seen as a set of pairs, each consisting of an output string and an associated weight. The identity element $\bar{0}$ serves as the weight of an empty path, whereas $\bar{1}$ serves as the neutral weight of a single transduction. In the rest of this section we define these concepts formally.

Definition 5 A ***finite-state machine*** M is a 9-tuple $M = (\Sigma_{in}, \Sigma_{out}, Q, i, c_i, F, C, E, (S, \oplus, \otimes, \bar{0}, \bar{1}))$ where:

Σ_{in} is a finite alphabet, called the input alphabet

Σ_{out} is a finite alphabet, called the output alphabet

Q is a finite set of states

[1] $\min(a,b) = \begin{cases} a \text{ if } a < b \\ b \text{ otherwise} \end{cases}$

$i \in Q$ is the initial state

c_i is the initial weight

$F \subseteq Q$ is the set of final states

$C : F \to S$ is the final weight function

$E \subseteq Q \times (\Sigma_{in} \cup \{\varepsilon\}) \times (\Sigma_{out} \cup \{\varepsilon\}) \times S \times Q$ is the set of transitions (arcs)

$(S, \oplus, \otimes, \bar{0}, \bar{1})$ is a semiring

Given a finite-state machine M = $(\Sigma_{in}, \Sigma_{out}, Q, i, c_i, F, C, E, (S, \oplus, \otimes, \bar{0}, \bar{1}))$ we define some additional notions. The ***state transition function*** $\delta : Q \times (\Sigma_{in} \cup \{\varepsilon\}) \to 2^Q$ of M is defined as:

$$\delta(p, a) = \{q \mid (p, a, b, s, q) \in E\} \tag{7}$$

The ***string emission function*** $\lambda : E \to \Sigma_{out}$ of M is defined by:

$$\lambda(t) = b \text{ for all } t \in E \text{ where } t = (p, a, b, s, q) \tag{8}$$

The ***weight emission function*** $\omega : E \to S$ of M is defined by:

$$\omega(t) = s \text{ for all } t \in E \text{ where } t = (p, a, b, s, q) \tag{9}$$

Definition 6 Let M be an FSM. A ***path*** $\pi = (t_1, t_2, ..., t_m)$ in M from q_0 to q_m is a finite sequence of transitions if for $i \in \{1,2,...,m\}$ $t_i = (q_{i-1}, a_i, b_i, s_i, q_i)$ and $q_i \in \delta(q_{i-1}, a_i)$. The first and the last state on the path π is represented by $start(\pi)$ and $end(\pi)$, resp.

Let $\pi = (t_1, t_2, ..., t_m)$ be a path. We now extend the notion of the string emission and weight emission function to paths:

$$\lambda(\pi) = \lambda(t_1) \circ \lambda(t_2) \circ ..... \circ \lambda(t_m) \tag{10}$$

$$\omega(\pi) = \bigotimes_{1 \le i \le m} \omega(t_i) = \omega(t_1) \otimes \omega(t_2) \otimes ..... \otimes \omega(t_m) \tag{11}$$

Further, we extend the notion of the string emission function to a set of paths as follows:

$$\lambda(C) = \bigcup_{\pi \in C} \lambda(\pi) \tag{12}$$

Definition 7 Let M be an FSM and $p, q \in Q$. With $p \xrightarrow{v} q$ we define the set of all paths from p to q with the input string v:

$$p \xrightarrow{v} q = \{\pi \mid \pi = ((p, a_1, b_1, s_1, q_1), ..., (q_{m-1}, a_m, b_m, s_m, q)) \text{is path in } M \text{ and } v = a_1 \circ a_2 \circ ..... \circ a_m\}$$

Given an input string $v \in \Sigma_{in}^*$, the set of all ***accepting paths*** can be defined as:

$$i \xrightarrow{v} F = \bigcup_{q \in F} i \xrightarrow{v} q \tag{13}$$

Since a given input string may correspond to different paths, which on the other hand produce different output strings, we would like to partition $i \xrightarrow{v} F$ into sets of paths, where all paths in a single set produce the same output string. In order to achieve this, we define an equivalence relation $SO \subseteq 2^{\Sigma_{in}^* \times E^* \times E^*}$ which reflects this property:

$$SO = \left\{ (v, \pi_1, \pi_2) \mid \pi_1, \pi_2 \in i \xrightarrow{v} F \text{ and } \lambda(\pi_1) = \lambda(\pi_2) \right\} \tag{14}$$

This relation yields an equivalence class:

$$[\pi_1]_v = \{ \pi_1 \mid (v, \pi_1, \pi_2) \in SO \} \tag{15}$$

The equivalence classes of SO form the desired partition:

$$\Pi_v = \left\{ [\pi]_v \mid \pi \in i \xrightarrow{v} F \right\} \tag{16}$$

Finally, we can define the output of an FSM.

Definition 8 Let *M* be an FSM. The ***output*** of *M* is a mapping $O_M : \Sigma_{in}^* \to 2^{\Sigma_{in}^* \times \Sigma_{out}^* \times S}$, where the output of *M* on the input string *v* is defined as follows:

$$O_M(v) = \bigcup_{P \in \Pi_v} \left(v, \lambda(P), \bigoplus_{\pi \in P} [c_i \otimes \omega(\pi) \otimes C(end(\pi))] \right)$$

The notion of rational transduction for FSMs could be adopted directly from definition 3. Analogously to FSAs, we can define the language accepted by an FSM. It is also called the ***domain*** of an FSM.

Definition 9 Let *M* be an FSM. The set of all input strings which are transformable by *M* is called the ***language accepted*** by *M* and is defined by:

$$L(M) = \{ v \mid \exists \pi \text{ in } M : \pi = ((i, a_1, b_1, s_1, q_1), ..., (q_{m-1}, a_m, b_m, s_m, q)) \\ \text{and } v = a_1 \circ a_2 \circ ..... \circ a_m \text{ and } q \in F \}$$

The set of all possible outputs of a given FSM is called the image of *M* and is defined as follows.

Definition 10 Let *M* be an FSM. The ***image*** of *M* is:

$$I(M) = \bigcup_{v \in L(M)} O_M(v)$$

Obviously, an FSM may accept an infinite number of input strings if there is an infinite number of paths in such FSM. This may occur when there exist cycles, which we define formally in the following definition.

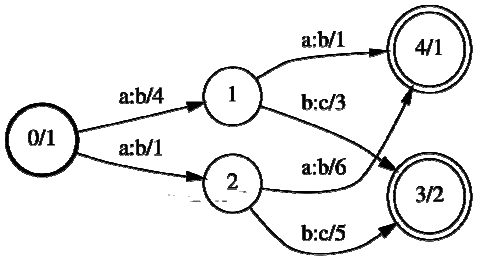
Definition 11 Let *M* be an FSM. A ***cycle*** in *M* is a path π with:

$$\pi = ((q_0, a_1, b_1, s_1, q_1), ..., (q_{m-1}, a_m, b_m, s_m, q_m)) \text{ and } q_0 = q_m$$

We extend the visual representation for FSMs as follows. The labels of the arcs contain input and output symbols separated by a colon and followed by a slash and the weight of the arc. Analogously, the labels of initial state and final states are extended with a slash followed by the weight associated with these states.

Example 3 Let $M = (\Sigma_{in\,M}, \Sigma_{out\,M}, Q_M, i_M, c_{i_M}, F_M, C_M, E_M, (R \cup \infty, \min, +, \infty, 0))$ be a finite-state machine which is defined as follows: $Q_M = \{0,1,2,3,4\}$, $F_M = \{3,4\}$, $\Sigma_{in\,M} = \{a, b\}$, $\Sigma_{out\,M} = \{b, c\}$, $E_M = \{(0, a, b, 4, 1), (0, a, b, 1, 2), (1, a, b, 1, 4), (1, b, c, 3, 3), (2, a, b, 6, 4), (2, b, c, 5, 3)\}$, $i_M = 0$, $c_{i_M} = 1$, $C_M(3) = 2, C_M(4) = 1$.

The FSM *M* accepts the language $L(M) = \{aa, ab\}$ and it transduces the string *aa* into *bb* and the string *ab* into *bc*. There are two paths corresponding to the input string *aa*, $\pi_1 = ((0, a, b, 4, 1), (1, a, b, 1, 4))$ and $\pi_2 = ((0, a, b, 1, 2), (2, a, b, 6, 4))$.



According to the definition 8 the output for the string *aa* is then:

$$O_M(aa) = (aa, bb, \min((c_{i_M} + \omega(\pi_1) + C_M(end(\pi_1))), (c_{i_M} + \omega(\pi_2) + C_M(end(\pi_2))))) = (aa, bb, \min(1+5+1, 1+7+1)) = (aa, bb, 7)$$

Analogously $O_M(ab) = (ab, bc, 9)$. Hence, $I(M) = \{(aa, bb, 7), (ab, bc, 9)\}$.

Definition 12 Let *M* be an FSM. *M* is a ***nondeterministic*** FSM, if and only if there exists a transition $t \in E$ with $t = (p, \varepsilon, b, s, q)$ and $p, q \in Q, b \in \Sigma_{out}, s \in S$ or there exist at least two transitions $t_1, t_2 \in E$ with $t_1 = (p_1, a_1, b_1, s_1, q_1)$, $t_2 = (p_2, a_2, b_2, s_2, q_2)$ and $p_1 = p_2$, $a_1 = a_2$. If these conditions are not fulfilled, then the FSM *M* is said to be ***deterministic***.

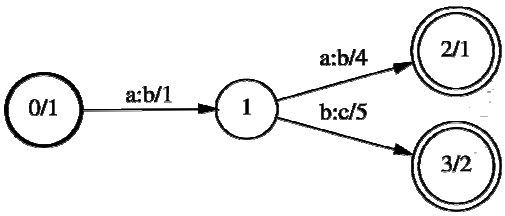


Figure 1. A deterministic FSM corresponding to the FSM in the example 3

The FSM in figure 1 accepts the same language as the FSM in example 3 and has the same image. Unfortunately, not all FSMs are determinizable [Mohri, 97]. Analogously to automata, there exist for every finite-state machine an equivalent FSM, which accepts the same language and has the same image and has the minimal size. The FSM in figure 2 corresponds to the FSM in example 3 and is minimal.

Definition 13 Let *M* be an FSM. An FSM $M_{\min}$ is ***minimal finite-state machine*** for *M* if $L(M) = L(M_{\min})$, $I(M) = I(M_{\min})$ and $|M_{\min}| \leq |M|$.

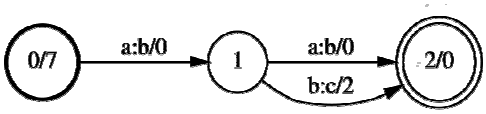


Figure 2. A minimal FSM corresponding to the FSM in the example 3

Lastly, we introduce the notion of useful states and useful FSMs.

Definition 14 Let *M* be an FSM. A state $q \in Q$ is said to be ***useful*** if and only if there exist $v \in L(M)$ and at least one path π, such that *q* lies on π and $\pi \in i \xrightarrow{v} F$. We say that an FSM is useful if all of its states are useful.

The notion of useful states is important since many of the algorithms implementing standard operations (e.g., intersection between two automata) produce as a result finite-state devices containing a huge amount of states which can never be reached or which do not lie on any path between an initial state and a final state. Moreover, some of the transformation algorithms yield proper results only when applied to useful finite-state devices.

An FSA or WFSA can be simply represented as an FSM which has identical input and output labels in each transition.

3. DFKI Finite-State Machine Toolkit

3.1 Introduction

The DFKI FSM Toolkit is a library of tools for constructing, combining, and optimizing the finite-state machines (FSMs). A Finite-state machine is a generalization of weighted finite-state automata (WFSA) and weighted finite-state transducer (WFST). The Toolkit provides all necessary functionalities relevant to the realization of different shallow text processing components (see [Piskorski and Neumann, 00]). Nevertheless, it embodies all major state-of-the-art finite-state operations and is designed with the consideration of future enhancements and applications, as well as its use within any finite-state based framework.

The architecture and functionality of the DFKI FSM Toolkit is mainly based on the tools developed by AT&T [Mohri et al., 00]. Analogously to the AT&T package, the finite-state machine model used here allows only a simple input/output symbol as a label of a transition and does not support final emissions. This is motivated by the fact that most of the algorithms require such types of transducers as input and thus time-consuming conversion steps may be omitted. Obviously, a finite-state machine that has output labels consisting of a sequence of output symbols could be easily transformed into a finite-state machine that has transitions on single symbols [Roche and Schabes, 96]. Further, final emissions could be easily simulated by introducing some additional transitions. Most of the provided operations are based on recent approaches, e.g., [Mohri et al., 97], [Roche and Schabes, 96], [Roche and Schabes, 97], [Daciuk, 98] and most of them work with arbitrary real-valued semirings (only computational representation of the semiring is needed). For instance, the tropical semiring can be used for finite-state pattern prioritization, whereas the real-valued semiring $(R_+, +, \cdot, 0, 1)$ is appropriate when dealing with probabilistic finite-state grammars[2].

The FSM tools are divided into two levels: an user-program level consisting of a stand-alone application that manipulates FSMs by reading from and writing to files and a C++-library level consisting of a library of C++ classes, methods and functions which implement the user-program level operations. A single operation from the user-program level is performed by a call to a corresponding C++ library function or method. This is preceded by a library function call to read each input FSM into an internal FSM object and followed by the library function call that writes out the resulting FSM. An example of performing an intersection of two finite-state machines is given in figure 3. The C++ library also includes classes that provide a public interface to the FSM object itself. The latter level allows an easy embedding of single elements of the toolkit into any other application by non-experts. Additionally, a JAVA interface to the FSM Toolkit is provided, which allows for writing JAVA applications which use optimized finite-state networks.

[2] If the weights represent probabilities, the weight assigned to a path should be the product of the weights of its transitions, while the weight assigned to a set of paths with a common source and target should be the sum of all path weights in the set.

user-program level:

fsm intersect in1.fsm in2.fsm out.fsm

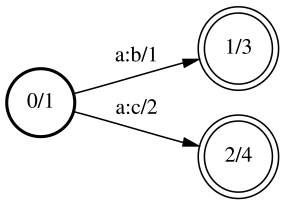
C++ library level:

```
FSM   in[2];
FSM   out;
in[0].initialize("in1.fsm");
in[1].initialize("in2.fsm");
out = fsm_intersection(in,2);
out.save("out.fsm");
```

Figure 3. An intersection of two finite-state machines realized on the user-program level and C++-library level

The DFKI FSM Toolkit supports two formats for representing FSMs: textual format, which eases converting FSMs to the textual format of some other finite-state packages, and compressed binary format, optimized for processing. In figure 4, a simple FSM together with its textual representation is presented.

The transitions of a finite-state machine are usually stored in so called transition tables [Hopcroft and Ullman, 79], also called ***transition matrix***. However, this requires $O(|Q| \cdot |\Sigma|)$ storage space, which might be seen as infeasible from the space complexity point of view. For compression of the transition table we adopted a variant of the row-indexed storage method presented in [Tarjan and Yao, 92] and [Wilhelm and Mauer, 92]. This method compresses the rows of the transition matrix into a single array A_M containing all transition information and an index array A_I with the information where the transition-entries for each state in A_M begin. A detailed description of this compression strategy is given in [Kiraz, 99].



(a)

```
0   1.0
0   1   a   b   1.0
0   2   a   c   2.0
1   3.0
2   4.0
```

(b)

Figure 4. (a) A simple FSM for converting *a* into *b* or *c* (b) textual representation

3.2 Overview of the Operations

In this subsection, the functionality of the FSM Toolkit is described in more detail. The operations of the FSM Toolkit are divided into four main pools: (a) converting operations, (b) rational and combination operations, (c) equivalence transformations, (d) auxiliary operations.

The group of converting operations includes converting textual representation into binary format and vice versa. Further, a conversion operation for creating graph representation in dot format from binary encoded FSMs is provided. The dot utility

[Koustofios, 96] developed at AT&T Bell Laboratories reads attributed graph text files and converts them into a graphics language such as Postcript.

The essential core algorithms are included in the pool of rational and combination operations and equivalence transformations. Algorithms on WFSAs have strong similarities with their better known unweighted counterparts, but the proper treatment of weights introduces some additional computations. On the other hand, algorithms on transducers are in general more complicated than the corresponding algorithms for automata. Since closure properties of transducers are much weaker than those of automata, some of the provided operations may only be applied to a restricted class of FSMs. Furthermore, only a small class of transducers is determinizable [Mohri, 97].

Apart from the standard operations some new operations are provided, which are of great importance in shallow text processing. Among others the new operations include direct incremental construction of minimal deterministic acyclic FSA, proposed in [Daciuk, 98], and local extension for WFSTs presented in [Roche and Schabes, 95]. The general algorithm for removing epsilon-transitions from weighted FSMs on arbitrary semirings has been modified. The group of rational and combination operations and equivalence transformations are described in more detail in subsection 3.3 and 3.4.

The auxiliary pool contains operations which are frequently used as subroutines in other operations. Nevertheless, they may be also used as stand-alone operations. This group includes for instance displaying statistical information about FSMs (e.g., number of epsilon transitions, number of nondeterministic moves), determinicity test, collecting arcs with identical labels and extending the input/output alphabet.

3.3 Rational & Combination Operations

The group of the rational and combination operations contains the following operations: (a) union, concatenation, Kleene-closure, inversion, reversion, complement, difference, and (b) composition, local extension, intersection.

The operations in (a) are well known [Roche and Schabes, 96], and since their implementation is straightforward, they will not be described. Just for the sake of clarity, the difference between inversion and reversion will be described here briefly. An ***inversion*** of an FSM M is an FSM M_{inv} where $\forall v \in L(M)$: $(v,u,c) \in O_M(v) \Leftrightarrow (u,v,c) \in O_{M_{inv}}(u)$. A ***reversion*** of a finite-state machine M is an FSM M_{rev} where $\forall v \in L(M)$: $(v,u,c) \in O_M(v) \Leftrightarrow (v^R,u^R,c) \in O_{M_{rev}}(v^R)$. The complement construction and difference operation are dedicated only to FSAs.

3.3.1 Composition

Composition is the main operation involved in the construction and use of transduction cascades [Pereira and Riley, 97]. The ***composition*** of an FSM M_1 and M_2 is an FSM $\hat{M}$ with following property: if M_1 transduces input string *s* to *t* with cost (weight) c_1 and M_2 transduces input string *t* to *v* with cost c_2, then $\hat{M}$ will transduce string *s* to *v* with costs $c_1 \otimes c_2$. The implementation of this operation is based on the algorithm proposed in [Pereira

and Riley, 97]. Note that the composition can only be constructed if the output alphabet of the first FSM is identical with the input alphabet of the second FSM.

3.3.2 Local Extension

Given an FSM that transforms *a* into *b* one would like to extend it to an FSM such that it transforms string *u* into *v*, where *v* is the word built from the word *u*, by replacing each occurrence of *a* by *b*. Such transformation of an FSM is called local extension. More formally a ***local extension*** of an FSM M is an FSM M_{locext}, which for all $u \in \Sigma_{in}^*$ with $u = x_1 y_1 x_2 y_2 .....y_{n-1} x_n y_n x_{n+1}$ and $\forall k \in \{1,....,n\}$: $y_k \in L(M)$ and $\forall l \in \{1,....,n+1\}$: $x_l \in \Sigma_{in}^* - (\Sigma_{in}^* \circ L(M) \circ \Sigma_{in}^*)$ and for all $(y_k, \bar{y}_k, c_k) \in O_M(y_k)$ transduces u into v where $v = x_1 \circ \bar{y}_1 \circ x_2 \circ ..... \circ \bar{y}_n \circ x_{n+1}$ with cost $c_1 \otimes c_2 \otimes ..... \otimes c_n$. Note that due to the definition above, the local extension of an FSM has to consider each possible factorization of the input string *u*.

The algorithm for computing local extension is taken from [Roche and Schabes, 95]. This has been extended in order to compute local extension of weighted finite-state machines. The local extension algorithm presented in [Roche and Schabes, 95] creates so called identity transitions for all symbols which can not be transformed at some state (merging many transitions into one transition). The FSM Toolkit enables the user to specify the symbol that will be used for representing identity transitions. Using this option obviously reduces the size of a resulting FSM. Since some operations on FSMs would not be able to interpret such transitions correctly, by default all identity transitions are expanded.

In the general case, a local extension of a given FSM is neither deterministic nor determinizable. Local extension together with composition are essential operations in the process of compilation of rewriting rules.

3.3.3 Intersection

Intersection is an operation usually used for merging single constraints on lexical or syntactic ambiguities in the context into a single finite-state device. Formally, an ***intersection*** [Roche and Schabes, 96] of two FSMs M_1 and M_2 is an FSM $M_\cap$ where $L(M_\cap) = L(M_1) \cap L(M_2)$ and $\forall v \in L(M_\cap)$: $(v, u, c_1) \in O_{M_1}(v)$ and $(v, u, c_2) \in O_{M_2}(v) \Leftrightarrow (v, u, c_1 \otimes c_2) \in O_{M_\cap}(v)$. Only $\varepsilon - free$ FSMs are closed under intersection. Hence, the ε symbol is treated as all other symbols while constructing intersection of FSMs.

3.4 Equivalence Transformations

The group of equivalence transformations include determinization, epsilon removal, trimming (removing states which are not useful), and bunch of minimization algorithms. The aim of these operations is to transform FSMs into deterministic, $\varepsilon - free$, useful and minimal FSMs, since processing such FSMs saves space and time.

3.4.1 Determinization

The determinization tool is based on the general determinization algorithm for arbitrary semirings presented by Mohri [Mohri, 97]. This algorithm constructs in the general case a *p*-subsequential transducer (see also 2.3), which allows up to *p* final emissions. They are represented in my FSM model by default by ε-transitions (reading nothing and emitting the symbols from final emissions). However, it is rather inconvenient to use ε-transitions, since there could also be some other outgoing ε-transitions from the same final state and the resulting FSM would not be deterministic in a strict sense. Further, localization of final emissions would be impossible in such case. Therefore, FSM Tools provide an option to specify what symbol (best choice is a symbol not included in the input alphabet) will be used as the input symbol in transitions which simulate final emissions. Nevertheless, the determinization algorithm usually constructs a deterministic FSM without final emissions when applied to FSMs used in real-world applications. In general, the determinization algorithm might not stop if the input is not determinizable since not all FSMs are determinizable. Note that WFSAs are determinizable only if they have the twins property [Buchsbaum et. Al., 00]. For the sake of clarity, we define it here briefly. If two states *q* and *p* are reachable from state *t* by a common string, then *q* and *p* are twins only if any string that induces a cycle at each, induces cycles of equal optimal cost. A WFSA is said to be have the twins property if all pairs *q* and *p* are twins (two states having no cycle on a common string are twins).

3.4.2 Minimization

Reducing the size of finite-state devices without losing their recognition and transition properties is crucial. Both finite-state automata and finite-state transducers have their minimal counterparts as described in section 2. For minimization of FSMs various algorithms are provided. In case of FSMs representing useful, unweighted and $\varepsilon - free$ FSAs the standard algorithm proposed in **[**Hopcroft and Ullman, 79**]** and **[**Watson, 94**]** is used. For minimization of their weighted counterparts a variant of the algorithm proposed in [Mohri, 94] is used. In the more general case of WFSTs, the well-known method proposed in [Brzozowski, 62] was used, which is capable of minimizing only a certain class of transducers, so called bideterminizable transducers, [Mohri, 97]. In this approach, a minimal FSM is obtained by applying determinization to the reversion of the input FSM, then reversing the obtained automaton and applying the determinization again. A transducer *T* is said to bideterminizable when its reversion *rev*(*T*) is determinizable and the reverse of *det*(*rev*(*T*)) can also be determinized. The standard algorithm for minimizing FSTs proposed in [Mohri, 94] has not been implemented since it is not suitable for letter transducers. Additionally, a novel and very fast utility for constructing a minimal, deterministic and acyclic finite-state automaton from the list of sorted words (or sorted sequences of words) is provided [Daciuk, 98]. It performs construction, determinization and minimization simultaneously in $O(m \cdot |w| \cdot \log|Q_{MIN}|)$, where Q_{MIN} denotes the state set of the minimized automaton, *m* is the number of words and $|w|$ is the maximum length of an input word. Due to its time complexity, this operation is crucial for on-line construction of gazetteers and encoding of other lexical resources.

3.4.3 Trimming

The operation of ***trimming*** transforms an input FSM into an equivalent useful FSM (an FSM with useful states) which accepts the same language and has the same image as the input FSM. Trimming is realized with a simple depth-first-search algorithm for graphs [Cormen et al., 92]. Removing inaccessible states and transitions is very useful since the FSMs returned by operations like intersection or composition generally contain many arcs and states that do not lie on any path from the initial state to a final state. Additionally, the trimming can be parametrized in order to remove symbols from an input/output alphabet which do not occur as input or output symbols on any arc of the input FSM. Removing such symbols may reduce the size of an FSM enormously.

3.4.4 Epsilon Removing

Epsilon removing is an essential operation in the process of determinization of FSMs and is usually performed directly before determinization. Further, many operations like for instance union or closure introduce epsilon transitions. For performing the task of removing epsilon transitions we adopted the general algorithm for arbitrary semirings presented in [Mohri et al., 96]. In contrary to FSAs, it is only possible to remove epsilon transitions for a restricted class of FSTs which have no ε-cycles. Due to the fact that FSMs allow only single symbols as input and output labels, the epsilon removal is restricted only to removing epsilon transitions of type ε/ε. However, the input FSM may contain ε-cycles if it represents a finite-state automaton (the type of input finite-state device is identified before the proper epsilon removal algorithm is triggered).

We will now briefly sketch the algorithm and introduce the modifications that proved to improve the performance of this algorithm when applied in the process of optimizing finite-state grammars used by the shallow processing system presented in [Piskorski and Neumann, 00]. The standard algorithm presented in [Mohri, 96] is divided into two phases. In the first phase, the input FSM *M* is subdivided into M_ε which contains only ε moves of *M*, and $\overline{M_\varepsilon}$ containing all other arcs. Subsequently, $\hat{M}_\varepsilon$ representing the transitive closure of M_ε is computed. Finally, in the second phase, the new equivalent $\varepsilon - free$ FSM is constructed from $\overline{M_\varepsilon}$ by iterating over the set of transitions of $\hat{M}_\varepsilon$ and performing appropriate weight modifications of existing arcs in $\overline{M_\varepsilon}$ and introducing new transitions to $\overline{M_\varepsilon}$.

Since the computation of the transitive closure in the general case of arbitrary semirings has complexity of $O(n^3)$ assuming that the computation of semiring operations $\otimes$ and $\oplus$ can be performed in $O(1)$ [Cormen at. al., 92], we propose two general modifications of the algorithm described above.

Firstly, in the preprocessing step all of the ***simple*** ε moves are removed from the input FSM, where an ε move is considered as simple when its target state does not have any outgoing ε arcs. Removing such transitions introduces occasionally new transitions to the input FSM and/or requires some weight modification of existing transitions, but since it is rather trivial, it will be not described here. Since the removing of simple ε transitions could yield some new ε transitions which are simple, this process is repeated until no more simple ε moves in the input FSM exist. Removing such transitions proved in practice to reduce the

overall number of ε moves significantly. The technique described here is more of a guideline for removing ε moves, since, depending on the input data, one could define simple ε moves differently and use various methods for removing them.

Secondly, analogously to the standard algorithm, the resulting FSM is then subdivided into M_ε and $\overline{M_\varepsilon}$. In the next step, the transitive closure of each connected component in M_ε is computed since one could expect them to be small in relation to the overall size of M_ε. The remaining procedure is identical to that of standard algorithm. Despite the fact that the modifications described here impair the worst-case time complexity, they proved to speed up the removal of ε moves considerably in the process of optimization of finite-state grammars in practice. In this way, we could experience that the formal complexity of algorithms might be significantly distant from the operational performance of algorithms. As a matter of fact, the second phase of the algorithm (computing the transitive closure) turned to be superfluous since there were no remaining ε moves.

3.5 Comparison to Other Similar Packages

This subsection presents a short comparison of the functionality of the FSM Toolkit with other similar finite-state packages. The figure 5 gives an overview of operations provided in the DFKI FSM-Toolkit, tools developed at AT&T [Mohri et. al., 00] and tools developed by van Noord [Noord, 98]. The symbol 'FSA' denotes that the operation is only available or applicable to finite-state automata. The Finite-state Tool from Xerox Parc [Kartunen et al., 96] has not been considered since it does not support operations on weighted finite-state devices.

The DFKI FSM-Toolkit was implemented in C++ and a corresponding extendible XML-based regular expression compiler with a user-friendly graphical interface is provided [Piskorski, et. al., 2002]. The AT&T Tools were written in C and show very good performance, whereas the Toolkit of van Noord was written in Prolog, and is equipped with user-friendly visualization tools for finite-state networks and a similar extendible regular expression compiler [Noord and Gerdemann, 99] which is not XML-based, but provides regular expression operators for regular relations. However, the latter tools do not provide operations for weighted finite-state transducers. Further, the AT&T package includes tools for computing finite-state approximations of context-free grammars [Mohri and Nederhof, 01] not present in DFKI FSM Toolkit. The main functional difference between the toolbox presented in this paper and the other tools is that DFKI Tools provide the operation for computation of local extension for weighted FSTs and an operation for incremental construction of minimal deterministic acyclic FSAs (denoted as perfect hashing in the table in figure 5), which are not provided in other packages. On the other hand best paths operations are missing, but their implementation is straightforward.

GFSMT – Generic Finite-State Machine Toolbox presented in [Scherf, 00] is another finite-state toolkit developed at DFKI. It was written in LISP, and offers nearly identical functionality as DFKI FSM Toolkit. However, it is rather dedicated for research purposes since the implementation was not efficiency oriented (e.g., transition tables are not compressed).

Other tools which provide some of the operations discussed in this paper were presented in [Daciuk, 00].

Operation	AT&T	FSA6	FSM Toolkit
UNION	+	+	+
INTERSECTION	+	+	+
CONCATENATION	+	+	+
CLOSURE	+	+	+
REVERSION	+	+	+
COMPLEMENT	FSA	FSA	FSA
DIFFERENCE	FSA	FSA	FSA
INVERSION	+	+	+
LOCAL EXTENSION			+
COMPOSITION	+	+	+
EPSILON REMOVAL	+	+	+
DETERMINIZATION	+	+	+
MINIMIZATION	+	+	+
EQUALITY	FSA		
TRIMMING	+	+	+
BEST PATHS	+	+	
PERFECT HASHING			FSA
VISUALIZATION	+	+	+
CFG-APPROXIMATION	+		
REGULAR COMPILER	+	+	+

Figure 5. Comparison of the functionality of DFKI FSM-Toolkit with finite-state tools developed by AT&T and tools of van Noord.

Acknowledgements

The research underlying this report and implementation of the FSM Toolkit was supported by a research grant from the German Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie (BMBF) to the DFKI projects: PARADIME, FKZ ITW 9704, and WHITEBOARD, FKZ: 01 IW 002.

I would also like to thank my colleagues from the Language Technology Lab in DFKI for fruitful discussions and valuable comments I received from them. In this respect, I am particularly indebted to Hans-Ulrich Krieger, Mark-Jan Nederhoff, Günter Neumann, Oliver Scherf, Wojciech Skut, Hans Uszkoreit, and Feiyu Xu.

Appendix A. A Quick User Guide to the DFKI FSM Toolkit

This is a quick guide to the user-program level of the DFKI Finite-State Machine Toolkit – a library of tools for constructing, manipulating, combining and optimizing weighted finite-state machines. We assume users familiarity with the theory of formal languages, and in particular regular languages and automata theory. The user-program level of FSM Toolkit provides two semirings for real numbers: (a) $(R \cup \infty, \min, +, \infty, 0)$ - the tropical semiring and (b) $(R_+, +, \cdot, 0, 1)$ - the 'plus-mul semiring'.

A.1. Representation of Finite-State Machines

DFKI FSM Toolkit uses textual and binary format for representing finite-state machines. Before carrying out any operation on an FSM represented in textual format, it must be converted into a compressed binary representation. Textual format consists of three files: FSM file (list of weighted transitions etc.), input alphabet file and output alphabet file, where the last two files are optional. If the input or output alphabet files are not explicitly defined, then the alphabets will be automatically extracted from the FSM file. However, in some applications it might be necessary to include some symbols in the alphabet even if they do not occur on any arc. This can be achieved only by using alphabet files. In textual format, each state and alphabet symbol is assigned an arbitrary string, whereas the string „EPS" is reserved for representing the empty string ε. Each arc cost is represented as a floating point number. The following is the textual representation of an FSM:

(1) the first line is of the form:

S C

where S is the initial state and C is the initial weight of the FSM.

(2) for each arc in the FSM, there is a line of the form:

S T IA OA C

where S is the source state, T is the destination state, IA is the input symbol, OA is the output symbol, and C is the arcs weight (last field is optional, if it is omitted, the arc is assigned weight 0.0).

(3) for each final state, there is a line of the form:

S C

where S is the state and C is the weight of accepting at that state. The second field is optional; if it is omitted, the state is accepted with weight 0.0.

The complete printed representation of an FSM consists of lines of the form in (1), (2) and (3). For example, an FSM represented by a file containing:

```
0    1.0
0    1    a    b    1.0
0    2    a    c    2.0
1    3.0
2    4.0
```

consists of 3 states, with an initial state 0, initial weight 1.0, two transitions (0 , a , b , 1.0 , 1) and (0 , a , c , 2.0 , 2) and two final states: 1 and 2 with final weights 3.0 and 4.0 respectively. In order to encode a finite-state automaton as an FSM, identity transitions should be used instead of using the empty string as an output symbol in the transitions.

The input/output alphabet file consists of a list of input/output symbols and their short names. For each input/output symbol there is a line of the form:

```
S    SN
```

where S is the input/output symbol and SN is a short name for S

In case an input/output alphabet file is used during converting textually represented FSM into binary format, there exist an option of extending all occurrences of short names to their full names if necessary. For example, using this option applied to the following input alphabet file:

```
Yellow      Y
Green       G
Blue        B
```

will cause all occurrences of input symbols Y, G or B in lines of type (2) in FSM file to be replaced by Yellow, Green and Blue during converting this FSM file into the corresponding binary representation.

A.2. Running the Toolkit

The user-program level of the FSM Toolkit consists of the stand-alone program **`fsm`**, which takes as input argument command name followed by the list of optional switches and other arguments depending on the command being used.

`General Syntax:` **`fsm command [switches] [arguments]`**

The switches may appear in an arbitrary order. In this section, all commands available in the current version will be described in more detail. Except the commands that compile and display information concerning FSMs, all the commands take as an argument one or more binary-encoded FSMs and send to files one binary-encoded FSM. The provided operations are divided into four pools: (a) converting operations, (b) rational and combination operations, (c) equivalence transformations and (d) auxiliary operations.

A.2.1. Converting Operations

COMPILE

SYNTAX: **fsm compile [options]** ***fsm_text fsm_bin***

DESCRIPTION: **compile** converts an FSM represented textually in the file ***fsm_text*** into compressed binary encoding and writes the result in the file ***fsm_bin***.

OPTIONS:

-w this option is required if the input FSM has weights.

-I ***ialph*** allows the inclusion of the input alphabet file stored in the file ***ialph***

-o ***oalph*** allows the inclusion of the output alphabet file stored in the file ***oalph***

-n if short names for input/output symbols have been used in transitions, then this switch has to be used in order to extend them properly.

PRINT

SYNTAX: **fsm print [options]** ***fsm_bin fsm_text***

DESCRIPTION: **print** converts binary-encoded FSM in the file ***fsm_bin*** into textual format and writes the result to the file ***fsm_text***.

OPTIONS:

-w this option has to be used in order to include the weights in the resulting FSM file in textual format.

-a creates an input and output alphabet file.

PRINT_INTERN

SYNTAX: **fsm print_intern** ***fsm_bin fsm_text***

DESCRIPTION: **print_intern** creates the compressed textual representation of the input FSM encoded in the binary file ***fsm_bin*** and writes the result to the file ***fsm_text***. This operation is used by the JAVA-Interface to FSM Toolkit, not described in this manual.

CV2DOT

SYNTAX: **fsm cv2dot *fsm_bin dot_txt***

DESCRIPTION: **cv2dot** converts the binary-encoded FSM ***fsm_bin*** into dot-format file ***dot_txt***. The **dot** utility developed at AT&T Bell Laboratories reads attributed graph text files and converts them into graphics language such as Postscript.

A.2.2. Rational and Combination Operations

Note that for the following operations the user may determine the semiring which will be used.

UNION

SYNTAX: **fsm union [options] *fsm_1 ... fsm_k fsm_res***

DESCRIPTION: **union** allows to build a union of two or more FSMs, where the input FSMs are stored in the files ***fsm_1,fsm_2,...,fsm_k*** and the result will be written to the file ***fsm_res***.

OPTIONS:

-s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

CONC

SYNTAX: **fsm conc [options] *fsm_1 fsm_2 fsm_res***

DESCRIPTION: **conc** computes the concatenation of two FSMs, where the input FSMs are stored in ***fsm_1*** and ***fsm_2***, and the result will be written to the file ***fsm_res***.

OPTIONS:

-s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

CLOSURE

SYNTAX: **fsm closure [options]** ***fsm fsm_res***

DESCRIPTION: **closure** builds the Kleene closure of the input FSM stored in the file ***fsm***. The resulting FSM is written to the file ***fsm_res***.

OPTIONS:

-e is used in order to exclude accepting of the empty string (Kleene **+** is used instead of Kleene *****).

-s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

INV

SYNTAX: **fsm inv [options]** ***fsm fsm_res***

DESCRIPTION: **inv** computes the inversion of the input FSM stored in the file ***fsm***, and writes the resulting FSM to the file ***fsm_res***.

OPTIONS:

-s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

REV

SYNTAX: **fsm rev [options]** ***fsm fsm_res***

DESCRIPTION: **rev** reverses the input FSM stored in the file ***fsm***, and writes the resulting FSM to the file ***fsm_res***.

OPTIONS:

-s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

COMPL

SYNTAX: **fsm compl [options]** ***fsm fsm_res***

DESCRIPTION: **compl** constructs the complement to the input FSM stored in the file ***fsm***, representing a deterministic, ε – *free* and cost-free finite-state automaton, and writes the resulting automaton to the file ***fsm_res***. The weights of all arcs and states of the resulting automaton are set to the neutral element of the extension operator of the currently used semiring.

OPTIONS:

–s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

DIFF

SYNTAX: **fsm diff [options]** ***fsm_1 fsm_2 fsm_res***

DESCRIPTION: **diff** computes the difference of two input FSMs representing finite-state automata stored in the files ***fsm_1*** and ***fsm_2*** respectively. The second FSM should represent a deterministic, ε – *free* and cost-free finite-state automaton. The result is written to the file ***fsm_res***.

OPTIONS:

–s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

COMPOSE

SYNTAX: **fsm compose [options]** ***fsm_1 fsm_2 fsm_res***

DESCRIPTION: **compose** builds the composition of two input FSMs stored in the files ***fsm_1*** and ***fsm_2***. The output alphabet of the first FSM must be identical with the input alphabet of the second FSM. Otherwise, the composition will not be constructed. The resulting FSM is written to the file ***fsm_res***.

OPTIONS:

-s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

LOCEXT

SYNTAX: **fsm locext [options]** ***fsm fsm_res***

DESCRIPTION: **locext** conctructs local extension of a given FSM stored in the file ***fsm***, and writes the resulting FSM to the file ***fsm_res***.

OPTIONS:

-I ***id_symbol*** defines the symbol which will be used for representing identity transitions (local extension algorithm creates so called identity transitions for all symbols in the input alphabet which can not be transformed at some state). By default all identity transitions are expanded to transitions with adequate symbols.

-s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

INTERSECT

SYNTAX: **fsm intersect [options]** ***fsm_1..fsm_k fsm_res***

DESCRIPTION: **intersect** returns the intersection of two or more FSMs, where the input FSMs are stored in the files ***fsm_1,...,fsm_k*** and the output FSM will be written to the file ***fsm_res***. The ε symbol is treated as all other symbols.

OPTIONS:

-s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

A.2.3. Equivalence Transformations

DET

SYNTAX: **fsm det [options]** ***fsm fsm_res***

DESCRIPTION: **det** turns the input FSM stored in the file ***fsm*** into an equivalent deterministic FSM and writes the result to the file ***fsm_res***. In general, this operation will not stop if the input FSM is not determinizable (note that WFSAs are determinizable only if they have twins property). Before determinizing FSMs representing finite-state automata, ε - transitions should be removed. The ε symbol is treated as all other symbols during determinization.

OPTIONS:

-f *fe_symbol* specifies what symbol will be used as input symbol in the transitions simulating final emissions (a symbol not included in the input alphabet would be a suitable choice) since the general determinization algorithm may produce an FSM containing final emissions. The final emissions are represented by default by ε - transitions (reading nothing and emitting the symbols from final emissions).

-s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

MIN

SYNTAX: **fsm min [options]** ***fsm fsm_res***

DESCRIPTION: **min** returns the minimal deterministic FSM equivalent to the to the input FSM stored in the file ***fsm***, representing either weighted bideterminizable transducer or weighted deterministic, ε – *free* finite-state automata. The result is written to the file ***fsm_res***.

OPTIONS:

-s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

MIN_ACC

SYNTAX: **fsm min_acc [options]** ***fsm fsm_res***

DESCRIPTION: **min_acc** returns the minimal deterministic FSM equivalent to the input FSM stored in the file ***fsm***, representing unweighted deterministic, ε – *free* and useful finite-state automaton. The result is written to the file ***fsm_res***.

OPTIONS:

-w *weight* is used in order to set all final weights in the resulting FSM to ***weight***.

RMEPS

SYNTAX: **fsm rmeps [options] *fsm fsm_res***

DESCRIPTION: **rmeps** removes all epsilon transitions of type ε/ε from a given FSM stored in the file ***fsm***. The input FSM may have ε-cycles only if it represents a finite-state automaton (removing of all ε/ε arcs from a finite-state transducer is not possible if it contains ε-cycles). The resulting FSM is written to the file ***fsm_res***.

OPTIONS:

-s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

CLEAN

SYNTAX: **fsm clean [options] *fsm fsm_res***

DESCRIPTION: **clean** returns an useful FSM corresponding to the input FSM stored in the file ***fsm***. The resulting FSM is written to the file ***fsm_res***.

OPTIONS:

-a allows to remove symbols from the input/output alphabet which do not occur as input or output symbols on any arc of the resulting FSM.

-s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

MINDET

SYNTAX: **fsm mindet [options] *input_file fsm_res***

DESCRIPTION: **mindet** constructs from a given list of multi-words stored in the file ***input_file*** the corresponding minimal deterministic finite-state automaton. The resulting FSM is written to the file ***fsm_res***.

OPTIONS:

-w *weight* is used in order to set all final weights in the resulting FSM to ***weight***.

-e is used for interpreting each word in a multi-word as a single symbol (e.g. 'New York' will be interpreted as two symbols in the input alphabet of the corresponding FSM). Note that in this case, all words in a multi-word must be separated by single space symbols.

A.2.4. Auxiliary Operations

LIST

SYNTAX: **fsm list**

DESCRIPTION: lists all currently available commands

INFO

SYNTAX: **fsm info [options]** ***fsm_bin***

DESCRIPTION: **info** prints information about the input FSM stored in the file ***fsm_bin*** to the standard output stream.

OPTIONS:

-e allows to obtain more detailed information about the input FSM (list of all transitions, final states etc.).

EXTALPHABET

SYNTAX: **fsm extalphabet [options]** ***a_file fsm fsm_res***

DESCRIPTION: **extalphabet** extends the input and output alphabet of the input FSM stored in the file ***fsm*** with the symbols listed in the alphabet file ***a_file***. The resulting FSM is written to the file ***fsm_res***.

OPTIONS:

-i *a_file* is used in order to extend only the input alphabet

-o *a_file* is used in order to extend only the output alphabet

COLLECT

SYNTAX: **fsm collect [options]** ***fsm fsm_res***

DESCRIPTION: **collect** merges identically labeled arcs (same input and output symbols, but different weights) between the same source and destination states of an input FSM stored in the input file ***fsm***. The resulting FSM is written to the file ***fsm_res***.

OPTIONS:

-s semiring specifies the semiring that is used. The parameter **semiring** may be set to ***tropical*** or ***plusmul*** for the tropical and real-plus semiring respectively. By default the tropical semiring is used.

IS_DETERMINISTIC

SYNTAX: **fsm is_deterministic [options]** ***fsm***

DESCRIPTION: **is_deterministic** carries out the determinicity test for the input FSM stored in the file ***fsm***. It returns the number of nondeterministic moves found in the input FSM to the standard output.

OPTIONS:

-f ***excl_symbol*** allows to exclude the symbol ***excl_symbol*** from being considered during the determinicity test (it seems to be useful to exclude the symbol used for denoting final emissions if the input FSM contains such transitions).

A.3. An Example of Using FSM Toolkit

As an introductory example of using FSM Toolkit let us consider a very simple case of constructing token classifier. For the simplicity, we only consider natural numbers. Let us assume that there are four classes of natural numbers: ‘one-digit number’, ‘two-digit number’, ‘four-digit number’ and ‘any natural number’. Obviously, these classes are not disjunctive. Further, let us assume that we want to assign each number class a priority in order to resolve potential ambiguities (e.g., ‘one-digit number’ is included in the class ‘any natural number’). We set the priorities as follows: ‘one-digit number’ - 1, ‘two-digit number’ – 2, ‘four-digit number’ - 3 and ‘any natural number’ - 4. In order to build corresponding number classifier represented as a single finite-state automaton, following steps must be undertaken:

(1) For each natural number class a corresponding finite-state automaton is created in textual format (***one-dig.txt***, ***two-dig.txt***, ***four-dig.txt***, **any-number*.txt***) and converted into compressed binary format. The priorities of the number classes are encoded in the final weights of the corresponding

automata (all final weights of the automaton representing the class ‘one-digit-number’ are set to 1).

```
fsm compile -w one-dig.txt one-dig.fsm
fsm compile -w two-dig.txt two-dig.fsm
fsm compile -w four-dig.txt four-dig.fsm
fsm compile –w any-number.txt any-number.fsm
```

(2) All automata created in the previous step are merged into a single automaton using the union operation. The use of tropical semiring seems to be suitable here. The resulting automaton is written to the file ***final.fsm***.

```
fsm union –s tropical one-dig.fsm two-dig.fsm
    four-dig.fsm any-number.fsm final.fsm
```

(3) The automaton created in the previous step is optimized by removing all epsilon transitions, determinization and minimization. The weighted minimal deterministic and epsilon free automaton representing the number classifier is now stored in the file ***final.fsm***.

```
fsm rmeps –s tropical final.fsm final.fsm
fsm det –s tropical final.fsm final.fsm
fsm min –s tropical final.fsm final.fsm
```

For access to more operations please refer to the user guide to the C++-library level of the FSM Toolkit or contact the author.

References

[Abney, 96] S. Abney. *Partial parsing via finite-state cascades*. In Proceedings of the ESSLLI 96 Robust Parsing Workshop, 1996.

[Ait-Mohtar and Chanod, 97] S. Ait-Mohtar, J-P. Chanod. *Incremental Finite-State Parsing*. In Proceedings of ANLP 97, Washington, USA, pp. 72-79, 1997.

[Aldezabel et. al, 01] I.Aldezabal, M.Aranzabe, A.Atutxa, K.Gojenola, M.Oronoz, K.Sarasola. *Application of Finite State Transducers to the Acquisition of Verb Subcategorization Information*. In: Finite State Methods in Natural Language Processing, Workshop at 13th European Summer School in Logic, Language and Information, Helsinki, Finland, 13-24 August, 2001.

[Black, 89] A. Black. *Finite State Machines from Feature Grammars*. In Proceedings of the International Workshop on Parsing Technologies, pages 277-285, 1979.

[Brill, 92] E. Brill. *A Simple Rule-Based Part-of-Speech Tagger*. In Proceedings of the Third Conference on Applied Computational Linguistics (ACL), Trento, Italy, 1992.

[Brzozowski, 62] J. A. Brzozowski. *Canonical regular expressions and minimal stage graphs for definite events*. Mathematical theory of Automata, 12:529-561, 1962.

[Buchsbaum et. Al., 00] A. Buchsbaum, R. Giancarlo, J. Westbrook. *On the Determinization of Weighted Finite Automata*. In Proceedings of SIAM Journal on Computing, 2000.

[Cormen et al., 92] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge., MA, 1992.

[Daciuk, 98] J. Daciuk. *Incremental Construction of Finite-State Automata and Transducers, and their Use in the Natural Language Processing*. Dissertation - Technical University of Gdansk, 1998.

[Daciuk, 00] J. Daciuk. *Finite-State tools for natural language processing*. In COLING 2000 Workshop on using Tools and Architectures to Build NLP Systems, pages 34-37, Luxembourg, August 2000.

[Daciuk, 01] J. Daciuk. *Computer-Assisted Enlargement of Morphological Dictionaries*. Finite State Methods in Natural Language Processing, Workshop at 13th European Summer School in Logic, Language and Information, Helsinki, Finland, 13-24 August, 2001.

[Daciuk and Noord, 01] J. Daciuk, G. van Noord. *Finite Automata for Compact Representation of Language Models in NLP*. Accepted for CIAA 2001.

[Eilenberg, 74] S. Eilenberg. *Automata, Languages, and Machines*. Volume A. Pure and Applied Mathematics: A series of Monographs and textbooks, Academic Press, New York, London, 1974.

[FSM 98] *Proceedings of the International Workshop on Finite State Methods in natural Language Processing*, Ankara, Turkey, June-July, 1998.

[Greffenstette, 96] G. Grefenstette. *Light Parsing as Finite-State Filtering*. In Kornai, András (ed.), Proceedings ECAI'96 workshop on Extended Finite State Models of Language, Budapest, Hungary, 1996.

[Gross, 89] M. Gross. *The Use of Finite Automata in the Lexical Representation of Natural language*. In Lecture Notes in Computer Science, 377, 1989.

[Hopcroft and Ullman, 79] J. Hopcroft, J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Series in Computer Science, Addison-Wesley, Reading, Massachusetts, Menlo Park, California, London, 1979.

[Kaplan and Kay, 94] R.M. Kaplan, M. Kay. *Regular models of phonological rule systems*. In Computational Linguistics, 20(3), 1994.

[Kartunen et al., 96] L. Kartunen, J-P. Chanod, G.Grefenstette, and A. Schiller. *Regular expressions for language engineering*. Natural Language Engineering, 2:2:305-328, 1996.

[Kartunen et al., 97] L. Kartunen, T. Gaal and A. Kempe. *Xerox Finite-state Tool*. Technical Report, Xerox Research Centre Europe, Grenoble, France, 1997.

[Kiraz, 99] G. Kiraz. *Compressed storage of sparse finite-state transducers*. In O. Boldt, H. Jürgensen, and L. Robbins, editors, Workshop on Implementing Automata WIA99 - Pre-Proceedings, Potsdam, July 1999.

[Koskenniemi, 83] K. Koskenniemi. *Two-level morphology: a general computational model for word-form recognition and production*. Technical Report: Publication No. 11., University of Helsinki Department of General Linguistics, 1983.

[Koustofios, 96] E. Koutsofios, S. C. North. *Drawing graphs with dot*. AT&T Bell Laboratories, Murray Hill, NJ, Technical Report, 1996.

[Krieger, 95] H-U. Krieger. A continuation-based ATN compiler. Technical Memo, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, 1995.

[Kuich and Salomaa, 86] W. Kuich, A. Salomaa. *Semirings, Automata, Languages*. EATCS Monographs on Theoretical Computer Science, Vol. 5, Springer, Berlin, 1986.

[Lewis and Papadimitriou, 81] H. Lewis, C. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[Mohri, 94] M. Mohri. *Minimization of sequential transducers*. Lecture Notes in Computer Science, 807, 1994.

[Mohri et. al., 96] M. Mohri, F. Pereira and M Riley. *A rational design for a weighted finite-state transducer library*. Technical report, AT&T Labs - Research, 1996.

[Mohri, 96] M. Mohri. *On some Applications of Finite-State Automata theory to Natural Language Processing*. Natural Language Engineering Vol. 2 No. 1, 2(1), pages 61-80, 1996.

[Mohri, 97] M. Mohri. *Finite-state transducers in language and speech processing*. Computational Linguistics 23, 1997.

[Mohri et. al., 00] M. Mohri, F. Pereira, and M. Riley. *The design principles of a weighted finite-state transducer library*. Theoretical Computer Science, 231:17-32, January 2000.

[Mohri and Nederhof, 01] M. Mohri and M. Nederhof. *Regular Approximation of Context-Free Grammars through Transformation*. In J. C. Junqua and Gertjan van Noord, editors, *Robustness in Language and Speech Technology*, pages 153-163. Kluwer Academic Publishers, Netherlands, 2001.

[Nederhof, 01] M. Nederhof. *Approximating Context-Free by Rational Transduction for Example-Based MT*. In Workshop Proceedings: Data-driven Machine Translation, at ACL/EACL, Toulouse, France, 2001.

[Noord and Gerdemann, 99] G. van Noord, D. Gerdemann. *An Extendible Regular Expression Compiler for Finite-State Approaches in Natural Language Processing*. Workshop on Implementing Automata, Potsdam, Germany, 1999.

[Noord, 00] G. van Noord. *FSA6 – manual*. Technnical report, http://odur.let.rug.nl/vannoord/Fsa/Manual/, Web document, 2000.

[Pavia, 99] N. G. Pavia. *Using the Incremental Finite-State Architecture to create a Spanish Shallow Parser*. In Proceedings of XV Congress of SEPLN, Lleida, Spain, 1999.

[Pereira and Riley, 97] F. C. N. Pereira, M. D. Riley. *Speech Recognition by Composition of Weighted Finite Automata*. In E. Roche and Y. Schabes, editors, *Finite-State Devices for Natural Language Processing*. MIT Press, Cambridge, Massachusetts, 1997.

[Piskorski and Neumann, 00] J. Piskorski, G. Neumann. *An Intelligent Text Extraction and Navigation System*. In the Proceedings of RIAO 2000 - Content-Based multimedia Information Access, Paris, France, 2000.

[Piskorski, et. al., 02A] J. Piskorski, W. Drożdżyński, F. Xu, O. Scherf. *A Flexible XML-based Regular Compiler for Creation and Converting Linguistic Resources*. In Proceedings of the 3rd International Conference on Language Resources an Evaluation (LREC) 2002, Las Palmas, Spain, 2002.

[Piskorski, et. al. 02B] J. Piskorski, T. Jäger, F. Xu. *A Framework for Domain and task Adaptive Named-Entity Recognition*. In Proceedings of the Fifth International Baltic Conference on Databases and Information Systems, Tallinn, Estonia, 2002.

[Revuz, 92] D. Revuz. *Minimisation of acyclic deterministic automata in linear time*. Theoretical Computer Science, 92(1): 181-189, 1992.

[Roche and Schabes, 95] E. Roche, Y. Schabes. *Deterministic Part-of-Speech Tagging with Finite-State Transducers*. Computational Linguistics, Vol. 21, Number 2, 1995.

[Roche and Schabes, 96] E. Roche and Y. Schabes. *Introduction to finite-state devices in natural language processing*. Technical report, Mitsubishi Electric Research Laboratories, TR-96-13, 1996.

[Roche and Schabes, 97] E. Roche and Y. Schabes. *Finite-State Language Processing*. MIT Press, Cambridge, MA, 1997.

[Siegel and Scherf, 00] M. Siegel, and O. Scherf. *Morphological Parsing of Japanese*. In: Conference Handbook of the Second International Conference on Practical Linguistics of Japanese. College of Humanities, San Francisco State University, 2000.

[Scherf, 00] O. Scherf. *GFSMT – General Finite-State Machine Toolbox*. Diploma Thesis, Computer Science Department, University of Saarland, Saarbrücken, Germany, 2000.

[Silberztein, 93] M. Silberztein. *Dictionnaires electroniques et analyse automatique de textes: le systeme INTEX*. Masson, Paris, France, 1993.

[Tarjan and Yao, 79] R. E. Tarjan, A. Yao. *Storing a sparse table*. In Communications of the ACM 22(11), 1979.

[Watson, 94] B. W. Watson. *A taxonomy of finite automata minimization algorithms*. Technical Report 93/44, Eindhoven University of Technology, The Netherlands, 1993.

[Wilhelm and Maurer, 92] R. Wilhelm, D. Maurer. *Übersetzerbau – Theorie, Konstruktion, Generierung*. Springer Verlag Berlin Heidelberg, 1992.

DFKI Finite-State Machine Toolkit

Jakub Piskorski

RR-02-04

Research Report