A Flexible XML-based Regular Compiler for Creation and Conversion of Linguistic Resources

Jakub Piskorski, Witold Drożdżyński, Oliver Scherf, Feiyu Xu

DFKI – German Research Center for Artificial Intelligence
Stuhlsatzenhausweg 3, 66 123 Saarbrücken, Germany
{piskorsk, witold, scherf, feiyu}@dfki.de

Abstract

Finite-state devices are widely used to compactly model linguistic phenomena, whereas regular expressions are regarded as the adequate level of abstraction for thinking about finite-state languages. In this paper we present a flexible XML-based and Unicode-compatible regular compiler for creating, and integrating existing linguistic resources. Our tool provides user-friendly graphical interface which enables the transparent control of the compilation process and allows for testing generated finite-state grammars with several diagnostic tools. Through the direct database connection, existing linguistic resources can be converted into user-definable finite-state representations.

1. Introduction

Finite-state devices are widely used to compactly model interesting linguistic phenomena because of their expressiveness and computational power. Regular expressions are regarded as the adequate level of abstraction for thinking about finite-state languages (Karttunen et al., 1996). They are usually used to encode linguistic resources like, for instance, definitions of token classes, context-dependent rewrite rules (e.g., part-of-speech filtering rules), grammars for the recognition of small-scale structures (e.g., nominal phrases, verb groups, named entities) and even high-level clausal patterns. Regular compilers are tools for converting regular expressions into their corresponding compressed finite-state representation. A variety of regular compilers based on advanced finite-state optimization toolkits have been presented in (Noord and Gerdemann, 1999; Karttunen et. al., 1997, Silberztein, 1997; Sproat, 1996) and has been successfully applied for the creation of linguistic resources in various domains of natural language processing, including phonology, morphology, part-of-speech filtering, shallow parsing and speech recognition.

In this paper we present a flexible XML-based regular compiler for creation, conversion and integration of existing linguistic resources. Both the definition of regular expressions and the configuration of the transformation process is done via XML which can be edited transparently in a user-friendly graphical editor. This allows for easy and straightforward extension (e.g., introduction of new operators and compilation options) and rapid processing by emerging technologies (e.g., XML parsers). The language resources engineer may flexibly define the way, in which the finite-state devices are merged together, and bias the optimization process. Further, distributed work is possible, since the compiler provides direct database access, which allows to convert existing linguistic resources stored in miscellaneous databases into user-definable finite-state representation in various formats. Additionally, diagnostic tools can be used to provide a deeper insight into the characteristics of the generated finite-state networks. Last but not least the compiler supports the Unicode encoding standard.

2. Regular Compiler

2.1. Architecture

The regular compiler consists of four main modules: (a) Graphical User Interface (GUI), (b) Compiler for regular expressions, (c) Finite-State Machine Toolbox, and (d) Driver for finite-state devices. The coarse-grained architecture of the tool is presented in figure 1.

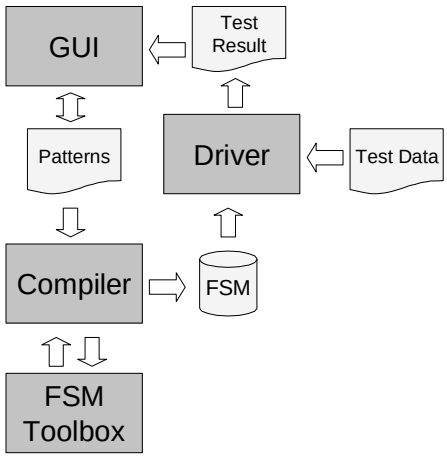


Figure 1. A coarse-grained architecture of the regular compiler.

Grammar development is done by simply encoding the regular patterns via XML, where the first part of the XML code contains compilation and transformation options, and the second part consists solely of pattern definitions. The user creates or modifies pattern definitions via the **GUI,** which is equipped with a kind of visual XML editor. Contrary to solely DTD or XML-schemata based editors, it has additional features helping the user to create valid XML.

Subsequently, the grammar is compiled. This invokes the **Compiler** module that takes the XML file generated by the GUI and validates it for consistency (e.g., the type of predicates in a certain scenario may be constrained). In

the next step, the patterns are converted into corresponding, optimized finite-state devices according to the defined compilation settings (e.g., type and output format of the generated finite-state devices, various optimization options). This task is fulfilled by an interaction with the **Finite-State Machine Toolbox** which is tightly coupled with the compiler. They exchange information in form of FSM objects or descriptions of them. Most of the regular operations are therefore realized as direct calls of the corresponding operations (or sequences of operations) of the FSM toolbox. Some of the compilation steps are done by the compiler. The result of the pattern transformation is then stored in one or more FSM files. Since FSM Toolbox constitutes the core component of the system, we describe it in some more detail in section 2.2.

Finally, the compilation results can be directly tested and explored via the **Driver** module. This includes testing the resulting FSMs on data provided by the user, and viewing their graphical representation and related statistical information. All results and information are uniformly visualized in the GUI.

The main advantage of the presented architecture is its modularity, since basic and important functionality is encapsulated as single modules. The FSM toolbox has a clear interface (FSM) and can be used as a standalone application or within other frameworks. The compiler takes XML files as input and can therefore be used without the GUI. Moreover, it can be combined with another finite-state package, since its interface is only based on an FSM model. Analogously, the driver can also be seen as a standalone component, since its interfaces are clear and well defined. Finally, the GUI may be deployed as a standalone component, because its XML processing is controled by a XML-based description and is not hard encoded. This allows for adapting its functionality very easily and quickly.

2.2. Finite-State Machine Toolkit

The Regular Compiler is based on the advanced Finite-State Machine Toolkit developed at DFKI (Piskorski, 2002), which provides efficiency-oriented implementation of state-of-the-art operations for constructing, combining and optimizing weighted finite-state machines which are generalizations of weighted finite-state automata and weighted finite-state transducers. The architecture and functionality of this toolkit is mainly based on tools developed by AT&T (Mohri, Pereira, and Riley, 1996). Analogously to the AT&T tools, the underlying finite-state machine model allows only single alphabet symbols as transition labels, since most of finite-state operations and transformations require this feature and therefore time consuming conversions may be avoided. The realization of most of the operations including equivalence transformations (e.g., determinization, removal of ε-transitions, minimization, and trimming) and combination operations (e.g., composition, intersection, union, difference, complement etc.) is based on the recent approaches proposed in (Mohri, 1997; Mohri, Pereira, and Riley, 1996; Roche and Schabes, 1996). Most of the provided operations work with arbitrary real-valued semirings.

In contrast to the AT&T tools, we provide some new operations relevant to NLP. For instance, the algorithm for local extension, which is crucial for merging part-of-speech filtering rules into a single finite-state transducer has been realized and adopted for the case of weighted finite-state transducers. Further, the toolkit provides an efficient operation for incremental construction of minimal deterministic acyclic finite-state automata from word lists proposed in (Daciuk, 1998), which is heavily used by the regular compiler. Finally, we also improved the general algorithm for removing ε-transitions (Mohri, Pereira and Riley, 1996) which is based on the computation of the transitive closure of the entire graph representing ε-moves. Instead of computing the transitive closure of the entire graph, we firstly remove all simple ε-transitions (a transition is considered simple if its target state does not have any outgoing ε-arcs) and then compute the transitive closure for each connected component in the graph representing the remaining ε-transitions.

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 and functions which implement the user-program level operations. The latter level allows for an easy embedding of single elements of the toolkit into any other applications. Currently, the FSM Tools run on UNIX, MS Windows and Linux platform. Detailed information concerning the tools can be found in (Piskorski, 2002).

2.3. Features

2.3.1. Regular Operators

The compiler provides circa 20 standard regular operators (Noord and Gerdemann, 1999). Some of the major regular operators are listed in figure 2. For the sake of clarity, we explain the semantics of the range, n-times and prefix operators. The operator n-times(*E,num*) corresponds to *num* concatenations of the regular expression *E* (E^{num}), whereas range(*E,min,max*) corresponds to the disjunction of E^{min}, E^{min+1}, …, and E^{max}. The operator prefix(*E*) corresponds to the concatenation of *E* with Σ*, where Σ denotes the alphabet used in the given set of patterns (suffix and infix are defined in an analogous way).

union(E_1,E_2,…,E_n)	concatenation(E_1,E_2,…,E_n)
star(*E*)	plus(*E*)
zero-one(*E*)	intersection(E_1,E_2)
complement(*E*)	difference(E_1,E_2)
range(*E,min,max*)	n-times(*E,num*)
prefix(*E*)	suffix(*E*)
infix(*E*)	sigma()
char-set(first,last)	

Figure 2. Basic regular operators.

New regular operators may be defined in terms of the existing ones, where such new operators are called functions. Let us consider as an example the definition of a simple regular-expression pattern for the recognition of date expressions in free-text document:

$[0\ldots9]^{1\text{-}2} \circ WS^{+} \circ [\text{'}January\text{'}, \ldots, \text{'}December\text{'}] \circ WS^{+} \circ [0\ldots9]^{4}$,

where *WS* denotes any whitespace. Since only month names have to be modified if one switches from one language to another, it would be convenient to define a generic parametrizable pattern which takes as an argument a list of month names. The XML definition of a corresponding general regular expression operator is presented in figure 3. The tag <ARG> denotes the argument of the operator, which is expected to be a regular expression representing names of the months, whereas the tag <PRE-DEF> denotes a call to a macro (e.g., 'Digit' is a macro representing a regular expression for recognition of digits).

```
<FUNCTION>
  <NAME> General Date Expression </NAME>
  <PARAM_COUNT> 1 </PARAM_COUNT>
  <REG-EXP>
    <CONCAT>
      <PRE-DEF>Digit</PRE-DEF>
      <ZERO-ONE><PRE-DEF>Digit</PRE-DEF></ZERO-ONE>
      <PLUS><PRE-DEF>WhiteSpace</PRE-DEF></PLUS>
      <ARG id="1">
      <PLUS><PRE-DEF>WhiteSpace</PRE-DEF></PLUS>
      <N-TIMES>
        <END> 4 </END>
        <REG-EXP>
          <PRE-DEF>Digit</PRE-DEF>
        </REG-EXP>
      </N-TIMES>
    </CONCAT>
  </REG-EXP>
</FUNCTION>
```

Figure 3. A one-argument regular operator representing a family of simple patterns for the recognition of date expressions

2.3.2. Importing Data from External Sources

External linguistic resources stored in databases or simple text files may be easily imported and converted into corresponding optimized finite-state devices. For instance, figure 4 shows a pattern for the recognition of date expressions in English, which is defined in terms of the general pattern for date expressions defined previously (see figure 3), where the list of the month names is imported from a textual file (the <INCLUDE> tag). In order to speed up the compilation process our tool also allows for importing regular expressions, which have previously been converted into optimized finite-state devices.

```
<PATTERN>
    <NAME> English Date Expression <NAME>
    <ID> 2 </ID>
    <REG-EXP>
      <FUNCTION>
        <NAME> General Date Expression </NAME>
        <ARGS>
          <INCLUDE>
            <FILE> month_names_English.txt </FILE >
          </INCLUDE>
        </ARGS>
      </FUNCTION>
    </REG-EXP>
</PATTERN>
```

Figure 4.A simple pattern for the recognition of English date expression

In order to import data from a database, an SQL-query must be formulated, where an additional user-defined pattern (combination pattern) determines how different fields of the database entries should be combined together into a desired format. In this sense, the tool may be used for merging existing linguistic data from miscellaneous sources into uniform compressed finite-state representations. An example of importing lexical entries from a database is presented in figure 5. The SQL-query is defined within the <SQL QUERY> tag (select all nouns with their corresponding stem and inflection information), whereas the code fragment tagged with <ROW_STRUCTURE> represents the combination pattern.

```
<INCLUDE>
  <DATABASE_ENTRY>
    <DB_WORD_LIST>
      <DB_NAME> VarLex </DB_NAME>
      <USER_NAME> UserName </USER_NAME>
      <SQL_QUERY>
          SELECT surface, stem, infl
              FROM lexicon1
                  WHERE POS='noun'
      </SQL_QUERY>
      <ROW_STRUCTURE>
          <DB_FIELD> surface </DB_FIELD>
          <SEPARATOR> : </SEPARATOR>
          <DB_FIELD> stem </DB_FIELD>
          <SEPARATOR> : </SEPARATOR>
          <DB_FIELD> infl </DB_FIELD>
      </ROW_STRUCTURE>
    </DB_WORD_LIST>
  </DATABASE_ENTRY>
</INCLUDE>
```

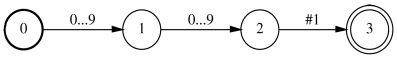
Figure 5. A fragment of a pattern definition for importing lexical entries from a database.

2.4. Compilation Settings

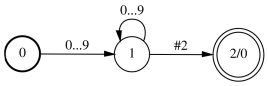
The compiler provides several options which determine the compilation process. First of all, the user may switch between ASCII or Unicode mode, which determines what character set may be used in the grammar writing process and simultaneously specifies the encoding style for the generated finite-state devices. Further, the user may decide whether the input data will be interpreted as scanner definitions (e.g., token class definitions) or general regular expressions. The finite-state devices generated by the compiler may be returned in various formats, including XML, compressed binary format optimized for processing, database entry or simple textual format, which enables to use the result in an arbitrary finite-state framework. Furthermore, there is an option of merging all patterns into a single optimized finite-state network instead of producing single finite-state devices for each defined pattern. The way in which patterns are merged may also be configured. For instance, the intermediate finite-state devices generated by the compiler may be optimized on demand (local optimization) in order to simplify global optimization.

Additionally, there are two alternative ways in which the ambiguities are handled. The first option is to resolve the ambiguities by assigning weights (<ID> tag in figure 4) to the patterns which represent their priorities (pattern prioritization). In the process of pattern merging, the

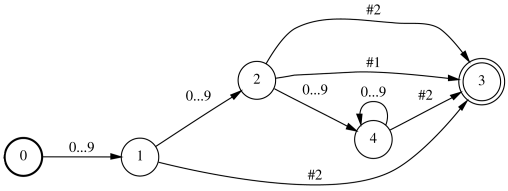
tropical semiring (Mohri, 1997) is applied in order to resolve potential ambiguities (i.e., the identifier of the matching pattern can be computed by combining the weight on the accepting path). The second option is to preserve all ambiguities by introducing appropriate final emissions representing pattern identifiers in the corresponding finite-state devices. The usage of the latter option is illustrated in figure 6 (the arcs labeled with the symbols "#1" and "#2" encode the pattern identifier).



(a) pattern for matching two-digit numbers



(b) pattern for matching all natural numbers



(c) automaton resulting by merging the automata in (a) and (b)

Figure 6. Merging finite-state devices representing ambiguous patterns.

A simplified example of compiler settings encoded in XML is illustrated in figure 7.

```
<COMPILER_SETTINGS>
  <UNICODE> true </UNICODE >
  <INPUT_SETTINGS>
    <INPUT_TYPE> scanner </INPUT_TYPE>
  </INPUT_SETTINGS>
  <OUTPUT_SETTINGS>
    <FILE>
      <FILE_FORMAT> xml </FILE_FORMAT>
      <OUTPUT_DIR> d:\tmp </OUTPUT_DIR>
      <EPSILON_CODE> EPS </EPSILON_CODE>
    </FILE>
    <SEPARATED_FSM> false </SEPARATED_FSM>
    <AMBIGUITIES> true </AMBIGUITIES>
  </OUTPUT_SETTINGS>
  <OPTIMIZATION_SETTINGS>
    <LOCAL_OPT> true </LOCAL_OPT>
  </OPTIMIZATION_SETTINGS>
<COMPILER_SETTINGS>
```

Figure 7. Compiler Settings

2.5. Graphical User Interface

Our tool is equipped with a graphical user interface shown in figure 8, which provides an intelligent XML editor. The user can switch between pure XML code view and tree view for defining patterns and configuration purposes. Both of the views can be modified. In the tree view no manual XML code writing is necessary, since the XML code is automatically generated by clicking buttons which correspond to all supported regular operators (user-defined operators too). Even for accessing special hard-to-find symbols, which have to be included in regular expressions, simple drag-and-drop mechanisms are provided (Unicode tables). At any point in the input window the user can choose new tags from a list of valid tags which obviously reduces potential mismatch errors.

The AT&T dot-Tool (Koustofios, 1996) was integrated in the GUI for the visualization of generated finite-state devices corresponding to single or merged set of patterns. Moreover, the patterns may be tested in a diagnostic window by using a simple finite-state driver and a text sample provided by the user (either ASCII or Unicode). The test results are displayed in an user-friendly way by the GUI, i.e., the part of the input that matches some pattern is highlighted and the list of all patterns that match with any prefix of the highlighted text passage may be displayed by simply clicking the mouse pointing at this marked passage (see figure 9). In this way, the user can easily and directly validate his pattern definitions, which facilitates grammar development. Various searching techniques and strategies may be chosen (e.g., longest matching vs. all matches, local vs. full backtracking, etc.).

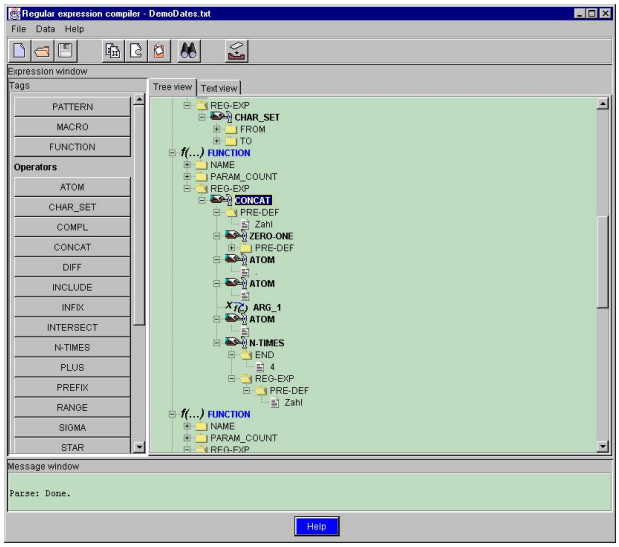


Figure 8. GUI for the Regular Compiler. The buttons on the left side are used for automatically generating XML code for all provided operators. The tree-like structure representing the XML code can be seen in the window on the right side (Tree view).

Finally, a tool for displaying detailed information about the characteristics of the resulting finite-state devices, including among others statistics about the number of nondeterministic moves, average number of outgoing transitions from a state, alphabet size etc. may support the language engineer in the process of optimally configuring the appropriate finite-state grammar interpreter and to ease the choice of an adequate storage model for the internal representation of the investigated finite-state device.

3. Application

Due to the user-friendly interface, parametrizable compilation and optimization options, Unicode compatibility, and various diagnostic tools, our tool proved to be an outstanding and reliable grammar development environment for multilingual pattern-based

shallow grammars and pattern-based template filler rules in the field of information extraction. In particular, we used this tool for the implementation of a small-scale structure recognition component for Chinese, and an information extraction subsystem applied in the management succession domain.

The major advantage of using the regular compiler in the process of Chinese grammar development (named-entity grammars) is the fact, that the compiler allows the user not only to enter Chinese signs from the keyboard, but also to access the signs from a Unicode table. An example of such table is given in figure 10. Moreover, we could easily convert various existing resources in Chinese (e.g. gazetteers) from different sources, which were useful for increasing the coverage of the discussed component.

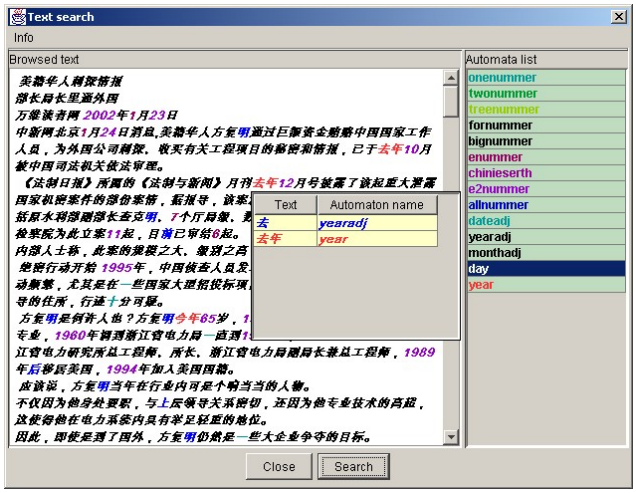


Figure 9. Diagnostic window for testing defined patterns.

In the second task we used the regular compiler for the definition of simple template filling rules for German person names which looks like follows:

'Successor' 'of' <PERSON NAME> → [Person_out: <PERSON NAME>]

This rule matches expressions like '*Successor of Helmut Kohl*' which contains two string tokens *Successor* and *of* followed by a person name, and fills the slot "person_out" with the recognized person name *Helmut Kohl*. Such rules are encoded in the compiler as regular expressions over the feature structures representing the output of the shallow analysis. The task-specific constraints (the regular compiler may be restricted to allow only predefined feature structures in the patterns) facilitate the grammar development process enormously.

4. Summary and Future Directions

In this paper we presented a flexible XML-based regular compiler for creation, conversion and integration of existing linguistic resources, which provides direct database connection, diagnostic tools, and a user-friendly graphical interface which enables to transparently control the compilation process. This tool was implemented in Java except the Finite-State Machine Toolkit, which was implemented in C++.

In order to improve the expressiveness capabilities of the tool we intend to integrate type definition system like TDL (Krieger, 1995), and extend the compiler by providing routines for the compilation of weighted context-dependent rules (Mohri, 2000) into corresponding optimized finite-state representations. Currently, we developed an extended experimental version of the compiler which handles unification-based regular grammars. Together with the intended enhanced driver capable of interpreting such grammars, the forthcoming version of the compiler will constitute an integrated grammar development and debugging environment for processing unification-based regular grammars. Finally, we aim at making the regular compiler publicly available and we also intend to perform usability test.

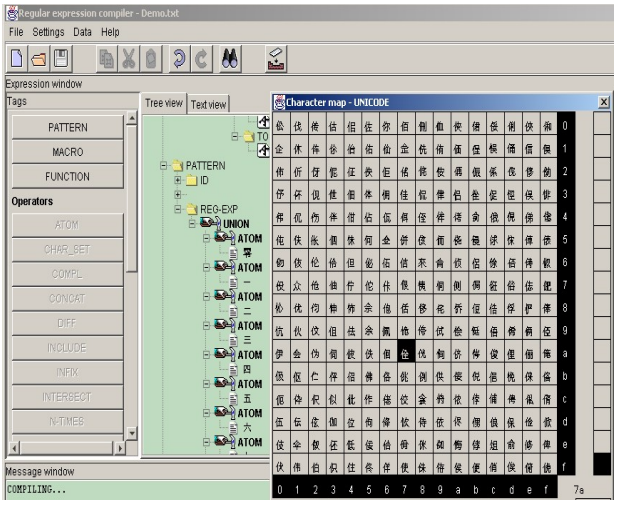


Figure 10. Unicode Table in action

5. References

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

Kaplan, R.M., Kay, Martin 94. *Regular Models of Phonological Rule Systems*. In Computational Linguistics, 20(3).

Kartunen, L., Chanod, J-P., Grefenstette, G., and Schiller, A., 1996. *Regular Expressions for Language Engineering*. Natural Language Engineering, 2:2:305-328.

Karttunen, L., Gaal, T., and Kempe, A., 1997. *Xerox Finite-state Tool*. Technical Report, Xerox Research Centre Europe, Grenoble, France.

Koustofios, E., North, S. C., 1996. *Drawing Graphs with Dot*. AT&T Bell Laboratories, Murray Hill, NJ, Technical Report.

Krieger, H.U., 1995. *TDL - A Type Description Language for Constraint-Based Grammars. Foundations, Implementation, and Applications*, Saarbrücken Dissertations in Computational Linguistics and Language Technology. Vol. 2.

Mohri, M., Pereira, F., Riley, M., 1996. *A Rational Design for a Weighted Finite-State Transducer Library*. Technical report, AT&T Labs - Research.

Mohri, M., Sproat, R., 1996. *An Efficient Compiler for Weighted Rewrite Rules*. In Proceedings of the 34th Annual Meeting of the ACL.

Mohri, M., 1997. *Finite-state Transducers in Language and Speech Processing*. Computational Linguistics 23.

Mohri, M., 2000. *Weighted Grammar Tools: The GRM Library*, in J.-C. Junqua and G. van Noord (eds), *Robustness in Language and Speech Technology*, Kluwer Academic Publishers.

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

van Noord, G., 2000. *FSA6 - manual*, TR, URL: http://odur.let.rug.nl/vannoord/Fsa/Manual/.

Piskorski, J., 2002. *DFKI Finite-State Machine Toolkit*, Technical Report, DFKI GmbH, Saarbrücken, Germany.

Roche, E., and Schabes, Y., 1996. *Introduction to Finite-State Devices in Natural Language Processing*. Technical report, Mitsubishi Electric Research Laboratories, TR-96-13, 1996.

Silberztein, M., 1997. *INTEX: A Finite State Transducer toolbox*. In Theoretical Computer Science 231:1, Elsevier Science.

Sproat, R., 1996. *Multilingual Text Analysis for Text-to-Speech Synthesis*. In the Proceedings of the 12th European Conference on Artificial Intelligence.