

FIREWood – Manual

R.G.W. Strolenberg L.G.W.A. Cleophas

February 15, 2009

Contents

1	Introduction	1
2	Installation	2
2.1	Source code based installation	2
2.2	Using the application	3
3	Basics	3
3.1	Specification tab	5
3.2	Automaton construction tab	5
3.3	Acceptance/Matching tab	7
4	Working with RTGs	8
4.1	Grammar transformations	9
5	Working with pattern sets	10
5.1	Automaton constructions	10
5.2	Matchers	10
A	FIREWood file format	10
A.1	Alphabets	11
A.2	Trees	12
A.3	Tree grammars	13
A.4	Tree patterns and pattern collections	14
A.5	Example	15

1 Introduction

FIREWOOD is a graphical application for experimenting with the FORESTFIRE toolkit. This toolkit contains a large collection of algorithms that focus on solving the tree acceptance and tree pattern matching problem, using tree or string automata. FIREWOOD therefore allows one to experiment with the various automaton constructions, acceptance/matching algorithms and other algorithms included in the toolkit.

The next section of this manual discusses the installation of FIREWOOD. This is followed by a section describing the application's user interface in general terms. The final two

sections focus on the specific possibilities for experimenting with respectively grammar transformations, automata constructions, and acceptors for tree grammars, and with automata constructions and pattern matchers for pattern sets.

2 Installation

FIREWOOD is implemented using Java, and distributed both as a source code package and an all-in-one JAR file. The standard procedure is to use the all-in-one JAR file, which is available for the Apple Mac OS X, Microsoft Windows XP, and Linux (with GTK 2.2.1 or newer) platforms. The source code package can be used to compile and generate new JAR files by hand when for instance extending FORESTFIRE and/or FIREWOOD. There are some prerequisites for both types of uses:

1. Precompiled JAR file
 - a) Java SE 5 or newer (<http://java.sun.com/>)
2. Compiling own JAR files
 - a) Java JDK 5 or newer (<http://java.sun.com/>)
 - b) SWT 3.3 or compatible version (<http://www.eclipse.org/swt/>)
 - c) Ant 1.7 or newer (<http://ant.apache.org/>)

To use a precompiled JAR file, read Section 2.2 named *Using the application*. To compile the source code yourself, read the next section.

2.1 Source code based installation

Compiling the source code of FIREWOOD can be done by using the ANT scripts contained in the source code package. Extracting this source code package results in the following directories:

- *ForestFIRE*, contains all FORESTFIRE source code.
- *FIREWood*, contains all FIREWOOD source code.
- *ForestFIREWood*, contains subdirectories with compilation related items:
 - *ant*, contains all Ant scripts for building both FORESTFIRE and FIREWOOD.
 - *dist*, used by Ant to store the resulting JARS.

There are two ANT scripts, one for compiling the source code to a single all-in-one JAR file containing FORESTFIRE, FIREWOOD, and the SWT library required by FIREWOOD (FW-BUILD-ONE.XML), and one to compile a modular version with separate FORESTFIRE, FIREWOOD JAR and SWT JAR files (FW-BUILD-MODULAR.XML). The scripts can be started using this command:

```
ant -Dswt=<path to swt.jar> -f <script file name>
```

After the compilation process, the ForestFIREWood/dist directory will either contain the all-in-one JAR file (FFW.JAR) or both the FORESTFIRE (FF.JAR) and FIREWOOD (FW.JAR) files with the additional SWT.JAR.

2.2 Using the application

The application can be started using a precompiled JAR file or the set of JARS created using the source code based installation. When using the all-in-one JAR file, FIREWOOD can be started using¹:

```
java -jar FFW.jar
```

When using the modular set of self compiled jar files one must ensure that these are in a single directory. FIREWOOD can then be started using the above command with FW.JAR instead of FFW.JAR.

When working with large input files or construction automata from large grammars or pattern sets it is wise to use the `-Xmx` parameter of JAVA (e.g. `-Xmx256m`) to allow the allocation of large amounts of memory¹:

```
java -Xmx256m -jar F(F)W.jar
```

3 Basics

When FIREWOOD is installed and started as described in Section 2 the user will see a large empty form (see Figure 1(a)). To start working with the application the user must create and load a FIREWOOD input file. Such a file may contain definitions of alphabets, trees, tree grammars, patterns and pattern sets. The structures in the file can be used to perform experiments inside the application, e.g. to create an automaton from a pattern set defined in the input file and use that automaton in a pattern matcher that solves the matching problem for a set of trees also taken from the input file.

There are two types of input files, plain text (that use the INI format) and compressed files (FZF extension). The format of the plain text files is described in Appendix A. An example file, defining three alphabets, one tree, one grammar, one pattern and one pattern set can be seen in Figure 1. The compressed files are ZIP archives containing a single plain text file (named after the FZF file's name but with INI extension). These zip files can be created by hand or by simply loading a plain text file into FIREWOOD and saving it as a compressed file.

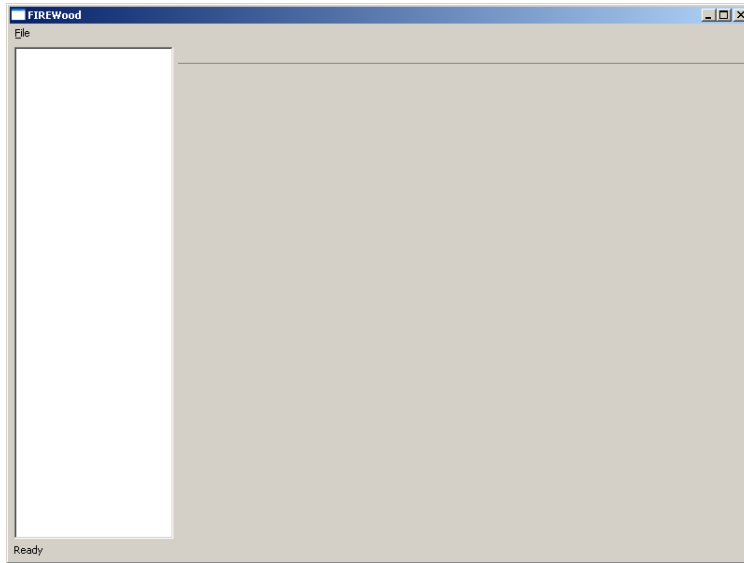
```
[TerminalAlphabet]
type=Alphabet
symbols={a:2, b:1, c:0, d:0}
```

```
[NonterminalAlphabet]
type=Alphabet
symbols={S, B}
```

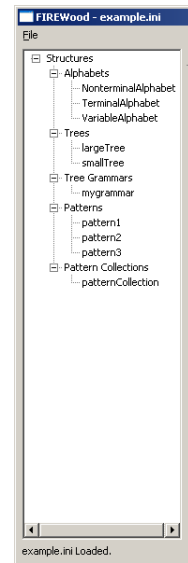
```
[VariableAlphabet]
type=Alphabet
symbols={v}
```

```
[smallTree]
type=Tree
alphabet=TerminalAlphabet
```

¹Apple Mac OS X users should additionally use the `-XstartOnFirstThread` parameter



(a) Empty main form



(b) Tree view after loading file

Figure 1: Main form, with and without loaded file

```

structure=a(b(c), d)

[mygrammar]
type=Grammar
terminal-alphabet=TerminalAlphabet
nonterminal-alphabet=NonterminalAlphabet
rules={B: S #0; B: b(B) #0; B: d #0; S: a(B, d) #0; S: a(b(c), B) #1; S: c #0}

[pattern1]
type=Pattern
terminal-alphabet=TerminalAlphabet
variable-alphabet=VariableAlphabet
structure=a(b(c),v)

[patternCollection]
type=PatternCollection
patterns={pattern1}

```

Code sample 1: Example plain text input file

Loading such an input file fills the tree view on the left of the main form with the structures defined in the file (see Figure 1(b)). A collection of operations is available for each of these structures, e.g. automata constructions for a tree grammar. These operations are accessible by selecting a structure in the tree view. Each time such a structure is selected a set of tab pages is opened in the right part of the application. Each tab page provides access to a specific set of operations, i.e. statistics and algorithms related to that type of structure. Figure 2, for example, shows the list of tab pages for a tree grammar. This list contains tab pages for analyzing the grammar, experimenting with transformations, building automata etc.

A subset of these tab pages is very similar for trees, tree grammars, pattern sets etc. A specification tab, containing the definition of the structure, can for instance be found for any

type of structure. Tree automaton construction pages are available for tree grammars and tree pattern sets, and are almost identical. The same holds for the acceptor tab for tree grammars and the matcher tab for pattern sets. The three categories of tabs are therefore discussed in respectively Section 3.1, 3.2 and 3.3. The particular details that differ between acceptance/grammars and pattern matching/pattern sets are discussed in Section 4 and 5 respectively. These two sections also discuss tab pages that are unique for these structures.

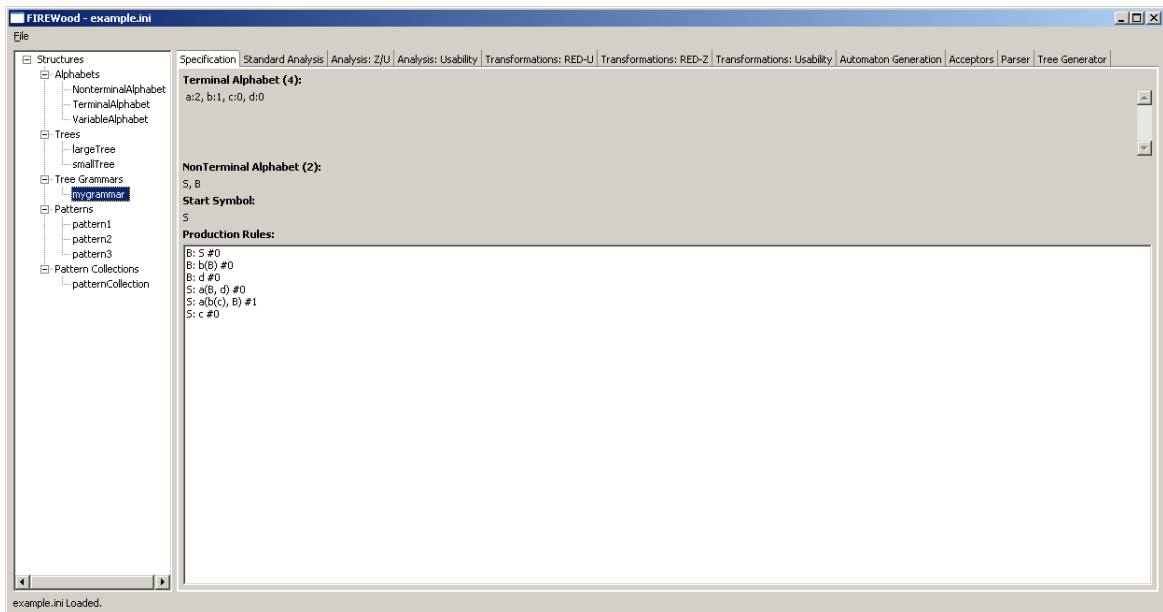


Figure 2: All tab pages for a tree grammar.

3.1 Specification tab

A specification tab is available for every structure and is always the first tab in the main tab control. The specification tab is designed to provide an overview of the definition of that structure. This overview is similar to the content of the plain text definition. For convenience, each reference inside a structure (e.g. an alphabet reference in a tree) is visualized by showing the structure to which the reference refers. Figure 2 shows this specification tab for a tree grammar, showing two alphabets instead of only the alphabet names/references. The list of production rules is similar to the list of rules in the input file. This tab often provides a clearer view of a structure than the input file itself, especially when working with large files.

3.2 Automaton construction tab

The FORESTFIRE toolkit contains many constructions resulting in different kinds of tree automata, for both tree grammars and tree pattern sets. All these constructions can be triggered using the automaton construction tab page. This tab page offers the possibility to construct an instance of each type of automaton with construction specific settings. The type of automaton can be chosen using a secondary, smaller tab control (see Figure 3), and the construction specific settings can be set using the controls on that secondary tab page.

After configuring the construction, the automaton can be built. Initially the automaton is

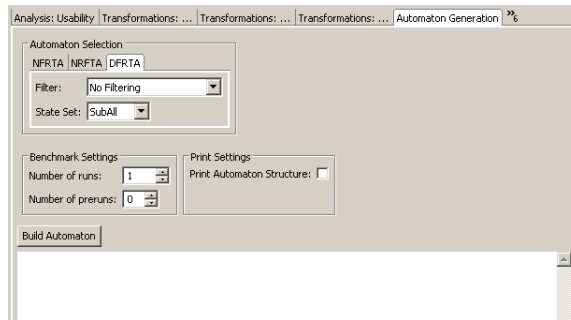


Figure 3: The automaton construction tab page for tree grammars

built and construction time, memory usage and automaton statistics are shown. The benchmark options can however be used to measure the average construction time over multiple constructions of the same automaton. The user can specify the number of runs that need to be performed before the measurements are started (to avoid negative influences of initial memory pool enlargements) and the number of constructions that should be used to determine the average running time. This way one can benchmark different automaton constructions.

Furthermore there is a possibility to print the internal structure of the automaton. When this is requested the automaton will provide a detailed plain text list of its states and transitions after all the construction cycles. The textual description of the automaton can then be used to study the structure of the automaton and compare it to other automata.

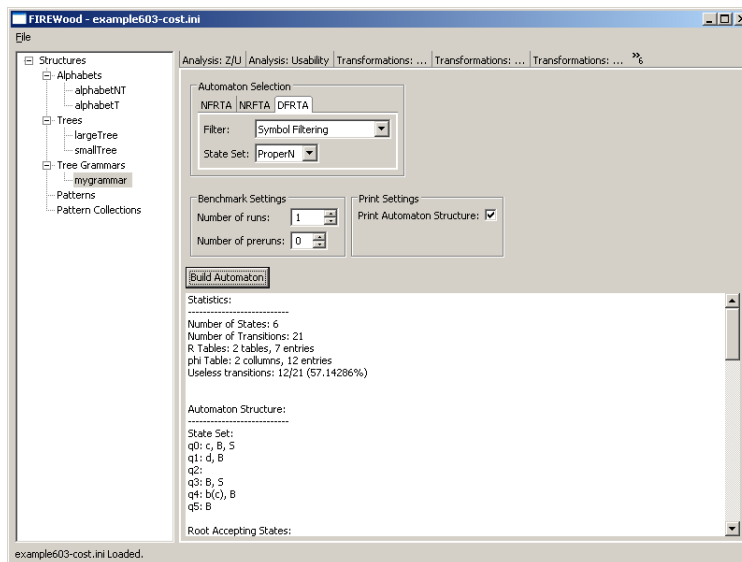


Figure 4: Result when building an automaton

To provide more clarity we provide an example of an automaton construction (for a tree grammar) using this tab page. The desired automaton will be a DFRTA with symbol filtering based on the PROPER-N item set. The measurements of this construction will be based on a single construction iteration with no iterations before the measurements are started. Figure 4 shows the settings and results of applying this construction. As described earlier one can just

print the statistics or both the statistics and structure of the automaton. Figure 4 shows both. Only printing the statistics would remove the list of states and transitions (partially shown). A user can use the output to learn more about the construction and the resulting automaton.

3.3 Acceptance/Matching tab

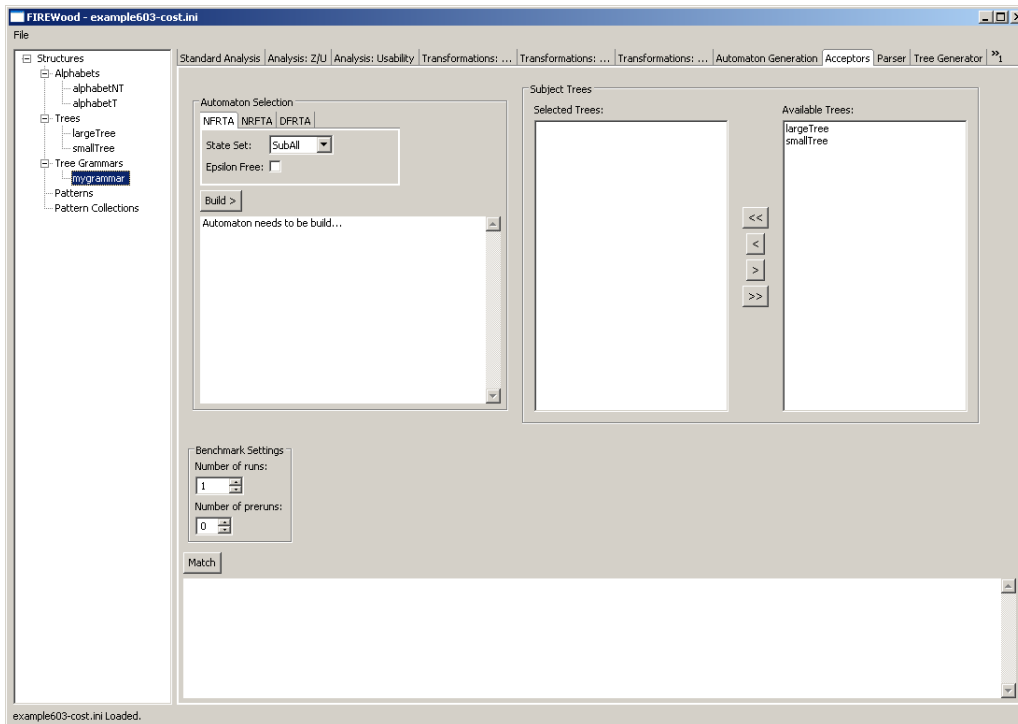


Figure 5: The tree grammar acceptor tab page

The tree acceptance and tree pattern matching algorithms implemented in FORESTFIRE are related to tree automata. Both kinds of algorithms take such an automaton and a set of trees and solve either the acceptance or matching problem for these trees using that automaton. The acceptors can be found in the acceptor tab for a selected tree grammar, and the pattern matchers in the matching tab for a selected tree pattern set. As there are more similarities than differences between many of the tabs for acceptance and matching, we discuss them as one here, deferring a discussion of the differences to Sections 4 and 5. The tab pages dedicated to acceptors/matchers (see Figure 5) start with an automaton selector comparable to the one on the automaton construction tab page. This automaton selector is a secondary, ‘subtab’ control on the tab page itself. Each time an automaton is built a small overview of its statistics is shown in the small text box below it. An automaton created with this control is also stored inside the subtab page, e.g. one can build a DFRTA, visit the NRFTA subtab page, and return to the DFRTA subtab page and finally trigger the acceptor that will then use the DFRTA. The acceptor/matcher tab will always use the automaton that is currently selected in the automaton selector.

Each type of automaton that can be selected in this control is coupled to a single type of

acceptor or matcher. This acceptor/matcher will then be created and applied on the trees the user has selected in the tree selector next to the automaton selector. This tree selector contains all trees specified in the input file.

Another group of controls present in the automaton construction tab page are the benchmark settings. For this tab page these are focussed at measuring the average running time of the acceptance/matching process for all selected trees over multiple iterations. Clicking the accept/match button will then start these iterations. The result (a boolean for acceptance, or a match set for each node when using a matcher) for each tree and the average running time for the iterations will be printed in the text box at the bottom. Figure 6 shows this output for the two trees in the input file using the DFRTA acceptor for the symbol filtered DFRTA based on the PROPER-N item set.

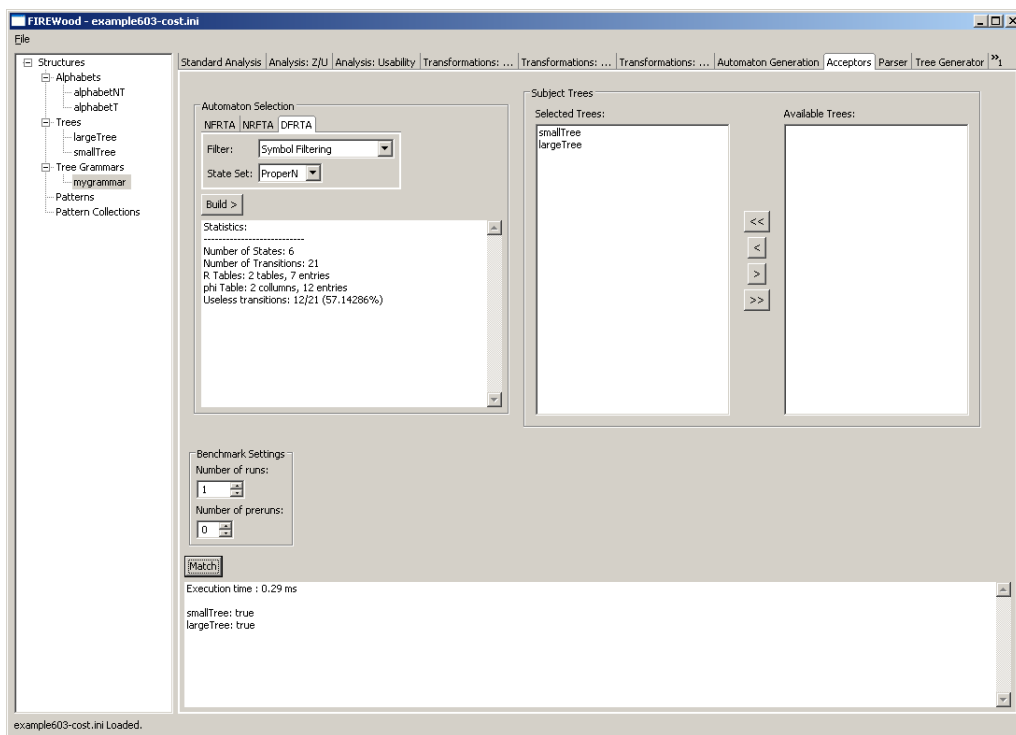


Figure 6: The tree acceptor output

4 Working with RTGs

FORESTFIRE implements a range of automaton constructions and acceptors for solving the tree acceptance problem for a given tree grammar and subject tree. These are accessible in FIREWOOD via the tab pages discussed in the previous section. FIREWOOD also contains tabs with operations for analyzing and transforming tree grammars:

- analyzing z-nodes and chain rules,
- analyzing useful and useless symbols and rules,
- experimenting with the RED-Z transformation,

- experimenting with the RED-U transformation, and
- experimenting with the usability transformations.

The tab pages related to transformations are discussed in Section 4.1.

4.1 Grammar transformations

There are three major grammar transformation algorithms in FORESTFIRE. These all have their own tab page. The transformation tab page for the RED-Z transformation will be highlighted in this section to give an overview of the possibilities for such a transformation. The two tab pages for RED-U and removal of useless symbols and rules are quite similar. Figure 7 shows an overview of the RED-Z transformation page for the grammar discussed in this chapter.

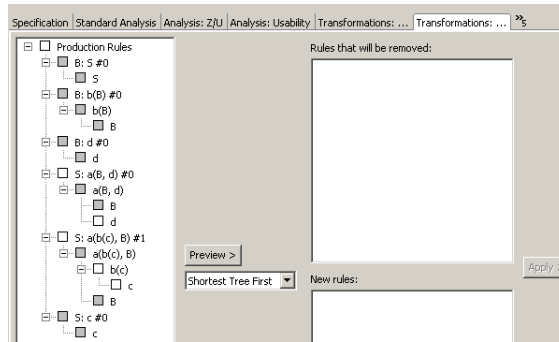


Figure 7: The RED-Z transformation tab page

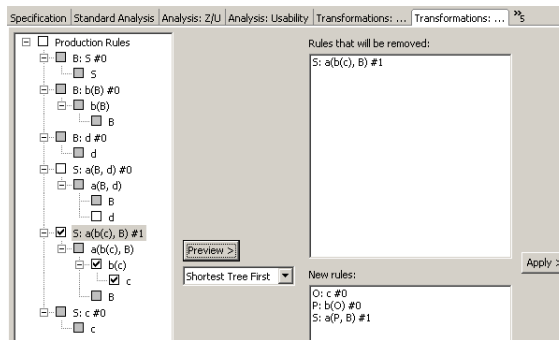


Figure 8: Transformation result preview

The left side of the tab page contains an overview of the rules of the selected tree grammar using a tree view. The z-nodes (non-root terminal nodes) in the production rules are selectable in this tree view. The user can then select the nodes he wants to remove using the transformation algorithm. Furthermore the user can use the combo box to select the desired node selection order for the transformation (see [2] and [1, Chapter 3]). Clicking the preview button will then show the result of this transformation in the two list views at the right. The top list shows the rules which will be removed and the bottom list shows the rules that are introduced to replace the removed rules (see Figure 8).

This way the user can see the effects of removing different sets of z-nodes with different strategies. Finally the user can apply this transformation to the grammar and store the resulting grammar using a new name.

5 Working with pattern sets

FORESTFIRE also contains a collection of algorithms for automaton construction for pattern sets and pattern matching using such automata.

In contrast to the tab pages for tree grammars there are only three for pattern sets: a specification tab, an automaton generator tab and a matcher tab. The automaton tab can, as expected, be used for experimenting with different automaton constructions that construct automata from pattern sets. The matcher tab is similar to the acceptor tab and can be used to experiment with different pattern matchers. The last two tab pages are discussed in more detail in the next two sections.

5.1 Automaton constructions

Automaton constructions can be performed via the automaton constructor tab page as described in Section 3.2. The tab page for pattern sets provides a broader choice than the one for tree grammars. It is possible to construct six different automata types: NFRTA, NRFTA, DFRTA, Aho-Corasick string path automata, SPDRFTA, and bit vector based automata (experimental and not documented yet—see source code for details).

5.2 Matchers

The tab page for matchers is exactly as described in Section 3.3. There are six types of automata available for tree pattern sets, and exactly these are available in the matching tab page. FORESTFIRE provides five types of matchers, where one of the matchers (string path matcher) works for both Aho-Corasick automata and SPDRFTAs. This tab page will select the matcher that can be used with the chosen automaton type and use that to solve the matching problem.

A FIREWood file format

This appendix discusses the FIREWOOD file format for defining trees, tree grammars etc. The goal of this file format is to provide an easy way to define input structures for the FIREWOOD application. The file format is based on the INI-format. We start by defining a *def-list*. This is a list of user defined structures. Each such a definition will be called a *def-item*.

These are five possible structures that can be defined:

- alphabet
- trees
- tree grammars
- tree patterns

- tree pattern collections

This resulted in the following EBNF-grammar for defining different structures:

```

def-list =
    def-item{def-item}

def-item =
    def-alphabet |
    def-tree |
    def-grammar |
    def-pattern |
    def-patterncollection

string =
    character{character}

character =
    a..z,A..Z

```

The definitions of the five different concepts have general shape, based on the general shape of an INI definition. Each definition of a concept consists of a name between brackets (the reference name of the concept) and a list of variable definitions, where one of the variable definitions specifies the type of the concept. This is for instance the general shape of a tree definition:

```

[mytree]
type=Tree
....=....
etc.

```

Each type of concept will contain a special list of variable definitions, next to the TYPE variable definition, that need to be present. A tree for instance needs to define its corresponding alphabet and its structure. See for instance the example specification in Section A.5. The next sections will discuss a format for each of the five different types of structures.

A.1 Alphabets

This section describes an alphabet definition. There were two possibilities for this definition. One could define one default alphabet for the complete file that every tree, tree grammar etc. uses or one could offer the possibility to define multiple alphabets. The last option was chosen, to provide the possibility of defining structures with different alphabets in a single file.

The list of variable definitions for the alphabet contains two elements: the type definition and the alphabet itself. The type must be set to ALPHABET and the alphabet itself is defined by a comma separated sequence of symbols and an optional rank (separated by the ':'-symbol). Where the symbols of the alphabet are strings of the characters a to z. Symbols with no rank are stored by ForestFIRE as unranked symbol. This is for instance used for nonterminals and variables.

This is the EBNF-grammar for this definition:

```
def-alphabet =  
    "[" string"]"  
    type=Alphabet  
    symbols=def-alphabet-list  
  
def-alphabet-list =  
    "{" def-symbol {,def-symbol}" }"  
  
def-symbol =  
    string [def-rank]  
  
def-rank =  
    ":" number  
  
number =  
    digit {digit}  
  
digit =  
    0..9
```

This is an example definition of a alphabet with name *alphabetx*:

```
[alphabetx]  
type = Alphabet  
symbols = {w:3, r:2, s:1, t:0, u:0}
```

A.2 Trees

Trees are defined by an alphabet and a definition of the tree structure itself. The alphabet is defined by a string that points to the defined alphabet with the same name. The tree structure is described in a prefix notation, where each node symbol is placed before a sequence of child nodes. This is the grammar of the tree definition, together with an example:

```
def-tree =  
    "[" string"]"  
    type=Tree  
    alphabet=string  
    structure=def-tree-structure  
  
def-tree-structure =  
    string | string "(" def-tree-children ")"  
  
def-tree-children =  
    def-tree-structure {,def-tree-structure}
```

This is an example definition of a tree with name *treex*:

```
[treex]
type = Tree
alphabet = alphabetx
structure = w(r(t,u),s(u),s(s(t)))
```

A.3 Tree grammars

Grammars are defined by a terminal alphabet, nonterminal alphabet, start nonterminal and a set of production rules. The terminal alphabet and nonterminal alphabet are defined in the same way as the alphabet of a tree by a string that points to a predefined alphabet. There is no separate part of the grammar definition that defines the start symbol. Instead the start symbol is defined as the first symbol in the nonterminal alphabet. This approach shortens the definition of a grammar.

The production rule list description is a bit more complicated. The rules are divided by semicolons and each rule is described by a string that represents the LHS followed by an ':' and a RHS tree structure as used in the tree definition. It is also possible to define a cost for each rule. This is done by putting a integer value after a '#' behind the production rule.

This is the grammar of the tree grammar definition, together with an example tree grammar definition:

```
def-grammar =
    "[" string "]"
    type=Grammar
    terminal-alphabet=string
    nonterminal-alphabet=string
    rules=def-productions

def-productions =
    "{" def-prule ; {def-prule} "

def-prule =
    string ":" def-tree-structure [def-cost]

def-cost =
    "#" number
```

This is an example definition of a tree grammar with name *grammarx*, together with an nonterminal alphabet called *alphabety*:

```
[alphabety]
type = Alphabet
symbols = {S, X, Y}

[grammarx]
type = Grammar
terminal-alphabet = alphabetx
nonterminal-alphabet = alphabety
rules = {S: w(X,Y,Y) # 1; X: r(t,X) # 3; X: u # 0; Y: s(Y) # 1; Y: t # 2; Y: X # 1}
```

A.4 Tree patterns and pattern collections

Let us start with the definition of tree patterns. The tree pattern definition is almost the same as the normal tree definition. There is only one difference. The pattern definition also contains an alphabet of variables. This alphabet is described just as a standard alphabet of tree, by pointing to the desired alphabet definition.

This is the grammar of the tree pattern definition together with an example of such a definition:

```
def-pattern =  
  "[ string ]"  
  type=Pattern  
  terminal-alphabet=string  
  variable-alphabet=string  
  structure=def-tree-structure
```

This is an example definition of a tree grammar with name *patternx*, together with an variable alphabet called *alphabetz*::

```
[alphabetz]  
type = Alphabet  
symbols = {v, w}  
  
[patternx]  
type = Pattern  
terminal-alphabet = alphabetz  
variable-alphabet = alphabetz  
structure = r(v,s(w))
```

Finally the pattern collection definition can be described. Such a collection is defined by the type 'PatternCollection' and a comma separated list of pattern variable names. These patterns should be defined in the same file before the definition of the pattern collection.

This is the grammar of this definition together with an example:

```
def-patterncollection =  
  "[ string ]"  
  type=PatternCollection  
  patterns=def-pattern-list  
  
def-pattern-list =  
  "{ string {, string } }"
```

This is an example definition of a tree grammar with name *patterncolx* with two imaginary names for the patterns:

```
[patterncolx]  
type = PatternCollection  
patterns = {patternx, patterny}
```

A.5 Example

This section shows the definition file for a tree grammar and a tree that can be produced by that grammar:

```
[alphabetT]
type=Alphabet
symbols={a:2, b:1 ,c:0, d:0}
```

```
[alphabetNT]
type=Alphabet
symbols={S, B}
```

```
[mytree]
type=Tree
alphabet=alphabetT
structure= a(b(c), b(b(d)))
```

```
[mygrammar]
type=Grammar
terminal-alphabet=alphabetT
nonterminal-alphabet=alphabetNT
rules={S : a (B, d); S : a (b(c), B); S: c; B: b(B); B: S; B: d}
```

References

- [1] Loek G. W. A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. PhD thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, April 2008.
- [2] Roger Strolenberg. *ForestFIRE & FIREWood, A Toolkit & GUI for Tree Algorithms*. Master's thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, June 2007.