Interactive Manipulation of Regular Objects with FAdo *

Nelma Moreira
DCC-FC& LIACC, Universidade do Porto
R. do Campo Alegre 823, 4150 Porto, Portugal
nam@ncc.up.pt

Rogério Reis
DCC-FC& LIACC, Universidade do Porto
R. do Campo Alegre 823, 4150 Porto, Portugal
rvr@ncc.up.pt

ABSTRACT

FAdo[1] is an ongoing project which aims the development of an interactive environment for symbolic manipulation of formal languages. In this paper we focus in the description of interactive tools for teaching and assisting research on regular languages, and in particular finite automata and regular expressions. Those tools implement most standard automata operations, conversion between automata and regular expressions, and word recognition. We illustrate their use in training and automatic assessment. Finally we present a graphical environment for editing and interactive visualisation.

Categories and Subject Descriptors

F.1.1 [**Computation by abstract devices**]: Models of ComputationAutomata; F.4.3 [**Formal Languages**]: Classes defined by grammars or automata; K.3.1 [**Computer Uses in Education**]: Computer-assisted instruction; K.3.2 [**Computer and Information Science Education**]: Computer science education

General Terms

Experimentation,Languages,Theory

Keywords

Automata theory, Regular languages, Interactive visual tools, e-learning

1. INTRODUCTION

Regular languages are fundamental computer science structures and efficient software tools are available for their representation and manipulation. But for experimenting, studying and teaching their formal and computational models it is useful to have tools for manipulating them as first-class objects. Automata theory and formal languages courses are mathematical in essence, and traditionally are taught without computers. Well known advantages of the use of computers in education are: interactive manipulation, concepts visualisation and feedback to the students. We believe that an automata theory course can benefit from this advantages, because:

- most of the mathematical concepts can be visualised graphically. Interactivity can help in the consolidation of the concepts and an easier grasp of the formal notation.
- most of the theorem proofs are algorithmic and can be interactively constructed
- automatic correction of exercises provides immediate feedback to the students, giving counter-examples and pointing out the errors, thus allowing for a quicker understanding of the concepts.

In this paper, we describe a collection of tools implemented in Python [11] that are a first step towards an interactive environment to teach and experiment with regular and other formal languages. The use of Python, a high-level object-oriented language with high-level data types and dynamic typing, allows us to have a system which is modular, extensible, clear, easy to implement, and portable. Python also provides several graphical and Web based libraries. Compared with Java language, it also has the advantage of an elegant syntax, and it is easy to learn, which makes it ideal for a first taught programming language.

In the next section, we describe the implementation of the core tools for regular languages symbolic manipulation. For more technical aspects, some familiarity with the Python language will be assumed. In Section 3 we show how **FAdo** can be used to solve and correct some automata theory problems. Section 4 briefly introduces a graphical environment and some interactive visualisations. Some related works are discussed in Section 5. Ongoing work is summarised in Section 6.

2. MANIPULATING REGULAR LANGUAGES

We assume basic knowledge of formal languages and automata theory [7]. The set of regular languages over an alphabet Σ contains $\emptyset$, $\{\epsilon\}$ where ϵ is the empty string,

*Work partially funded by Fundação para a Ciência e Tecnologia (FCT) and Program POSI.

[1]**FAdo** is an acronym of *Finite Automata devoted oracle* and also a genre of portuguese popular music; the project page is `http//www.ncc.up.pt/fado`.

ITiCSE'05, June 27–29, 2005, Monte de Caparica, Portugal.

$\{a\}$ for all $a \in \Sigma$, and is closed under union, concatenation and Kleene closure. Regular languages can be represented by regular expressions (`regexp`) or finite automata (`FA`), among other formalisms. Finite automata can be deterministic (`DFA`) or non-deterministic (`NDFA`). All three notations can represent the same set of languages. In **FAdo**, we can manipulate each of these representations and convert between them, as shown in Figure 1.

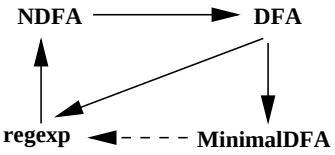


Figure 1: Conversions between regular language representations

2.1 Finite Automata

Formally a deterministic finite automaton is specified by a 5-tuple $(S, \Sigma, \delta, s_0, F)$, where S is the set of states, Σ is the input alphabet, δ is the transition function $\delta : S \times \Sigma \to S$, s_0 the initial state, and $F \subseteq S$ is the set of final states. In a nondeterministic automata δ is a function from $S \times \Sigma$ to the set of subsets of S ($\mathcal{P}(S)$), $\delta : S \times \Sigma \to \mathcal{P}(S)$.

The class `FA` implements the basic structure of finite automata shared by deterministic and non-deterministic ones. This class also provides methods for manipulating these structures: add, set, delete, test, etc. A list of its main attributes and methods can be found in Table 3, Appendix A.

2.1.1 Nondeterministic Automata

The class `NDFA` inherits from the class `FA`, and provides methods to manipulate a `NDFA`. In the literature, there is a distinction between `NDFA` with and without ϵ-transitions (`NDFA` and ϵ-`NDFA`). In **FAdo**, we allowed all `NDFA`'s to be ϵ-`NDFA`. But we provide methods to test for ϵ-transitions and to convert an ϵ-`NDFA` to a `NDFA`. See Table 4, Appendix A, for more details.

2.1.2 Deterministic Automata

The class `DFA` inherits from the class `FA`, and provides methods to manipulate a `DFA`. Mathematically `DFA`'s are richer than `NDFA`'s. In the next paragraphs we analyse some of those features.

A Canonical Form for `DFA`'s.

It is possible to test if two `DFA`'s are equivalent, and given a `DFA`, to find an equivalent `DFA` that has a minimum number of states. The method `Minimal()` implements `DFA` minimisation using the *table-filling* algorithm [7]. For testing equivalence of two `DFA`'s, we can minimise the two automata and verify if the two minimised `DFA`'s are *isomorphic* (i.e are the same up to renaming of states). For verify isomorphism we developed a canonical form for `DFA`'s. Given a `DFA` we can obtain a unique string that represents it. Let Σ be ordered (p.e, lexicographically), the set of states is reordered in the following manner:the initial state is the first state; following Σ order, visit the states reachable from initial state in one transition, and if a state was not yet visited, give it the next number; repeat the last step for the second state, third state, ... until the number of the current state is the total number of states in the new order. For each state, a list of the states visited from it is made and a list of these lists is constructed. The list of final states is appended to that list. The result is a *canonical form*. If a `DFA` is minimal, the alphabet and its canonical form uniquely represent a regular language. For test of equivalence it is only needed to check whether the alphabets and the canonical forms are the same, thus having linear costing time.

Other `DFA` Operations.

Regular languages are also closed under other operations, such as intersection, complement, difference of two languages and reverse. In the core implementation we choose to define, in the class `DFA`, those operations which are closed for `DFA`'s (we excluded, concatenation and Kleene closure). See Table 5, Appendix A, for more details.

Producing a Witness of the Difference of two `DFA`'s.

Sometimes it is useful to generate a word recognisable by an automaton. This is the case in correcting exercises where we have the solution and a wrong answer from a student. Instead of a simple statement that an answer is wrong, we can exhibit a word that belongs to the language of the solution, but not to the language of the answer (or vice-versa). A *witness* of a `DFA`, can be obtained by finding a path from the initial state to some final state. If no *witness* is found, the `DFA` accepts the empty language. Given `A` and `B` two `DFA`'s, if $\neg$`A`$\cap$`B` or `A`$\cap\neg$`B` have a witness then `A` and `B` are not equivalent. If both `DFA`'s accept the empty language, `A` and `B` are equivalent. This test is implemented by the method `witnessDiff()`.

2.1.3 Converting `NDFA`'s to `DFA`'s

The equivalence of nondeterministic and deterministic automata is one of the most important facts about regular languages. Trivially a `DFA` can be seen as a `NDFA`. The conversion of a `NDFA` to a `DFA` that describes the same language, can be achieved by *subset construction* [7]. This method is usually taught in automata theory courses, though its illustration and animation are very useful. It is implemented by the module function `NDFA2DFA()`.

2.1.4 File Format for I/O

We have a very simple format to read and write finite automata definitions. Each file can contain several definitions and must obey the following specifications. An `#` begins a comment. An `@DFA` or `@NDFA` begins a new automata (and determines its type). It must be followed by the list of the final states separated by blanks. Each following line represents a transition. It is a triple that consists of the source state (name), an input symbol, and the target state (name). Each of the fields are separated by a blank. The name of a state can be any string (except `#`). Finally, the source state of the first transition is the name of the initial state.

Besides the methods for reading and writing in this native format, we also have some export/import filters for standard file formats, e.g., an extension of GraphML [3] and Dot Language [4].

2.2 Regular Expressions

A regular expression can be a symbol of the alphabet, the empty set ($\emptyset$), the empty string (ϵ) or the concatenation or the union ($+$) or the Kleene star ($\star$) of a regular expression.

The class `regexp` implements the three base cases and the

complex cases are the subclasses `concat`, `disj` and `star`, respectively. The constant `Epsilon` represents the empty string and the constant `Emptyset` represents the empty set. See Table 1 and Table 2, Appendix A, for more details.

Equivalence and simplification of regular expressions is a major topic of research in automata theory. Minimal equivalent expressions are not unique and no canonical form is known. Although, complete axiomatizations can be used to test equivalence and perform simplifications, we currently implemented very naive simplification rules and conversion of regular expressions to `DFA`'s (see Section 2.3). Nevertheless, one of our future goals is to implement rewriting systems to perform regular expressions simplifications.

2.3 Converting Finite Automata to Regular Expressions

The standard conversion from `DFA`'s to regular expressions, is based on successively constructing regular expressions $r_{ij}^{(k)}$, that represent the language recognised between state i and state j, without going through a state number higher than k [7]. This algorithm is implemented by the method `regexp()` of the class `DFA`. This algorithm is mathematically very instructive, but it is highly inefficient. So we also implemented a less redundant method of elimination of states [7]. This algorithm is also easily animated, and, in the **FAdo** graphical interface it is possible to choose, in each step, which state to eliminate.

2.4 Converting Regular Expressions to Finite Automata

The basic conversion is from regular expressions to ϵ-`NDFA`'s using the *Thompson's* construction [7]. The idea is to recursively build an ϵ-`NDFA` for each type of `regexp`. Each `regexp` subclass has a method `ndfa()` that allows to construct an `NDFA` for its type. Due to its applications in pattern recognition, this conversion is much studied in the literature and many algorithms are known. In particular, the Glushkov construction [16], is currently implemented in **FAdo** in a separated module.

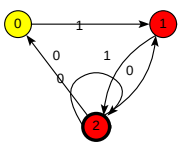
3. USING FAdo

In this section we illustrate how **FAdo** can be used for solving (and correcting) some typical exercises for a first course in automata theory. In Python interactive mode, we must first import the package:

```
>>> from dfa import *
```

EXAMPLE 1. *Convert to a regular expression the following* `NDFA`*:*

```
@NDFA 2
0 1 1
1 1 2
2 0 1
2 0 2
2 0 0
```




```
>>> n=readFromFile("examples/e1.fa")[0]
>>> d=NFDA2DFA(n)
>>> print d.regexp().simplify()
((11)+(11))+(1100*1((0+(10))0*1)*(1+1))
```

EXAMPLE 2. *Convert the regular expression* $(0+1)^\star(012)$ *to a* `NDFA`*. Obtain an equivalent minimal* `DFA` *and the* `DFA` *canonical form.*

```
>>> a=str2regexp("(0+1)*(012)")
>>> d=NDFA2DFA(a.ndfa())
>>> d.Minimal()
>>> saveToFile("e2.fa",d)
>>> d.uniqueStr()
[[1,0,2],[1,3,2],[2,2,2],[1,0,4],[2,2,2],[4]]
```

```
@DFA 3
0 1 0
0 0 1
0 2 4
1 1 2
1 0 1
1 2 4
2 1 0
2 0 1
2 2 3
3 1 4
3 0 4
3 2 4
4 1 4
4 0 4
4 2 4
```

EXAMPLE 3. *Check whether the following two regular expressions are equivalent* $(01+0)^\star$ *and* $0(10+0)^\star$*.*

```
>>> a=str2regexp("(01+0)*")
>>> b=str2regexp("0(10+0)*")
>>> da=NDFA2DFA(a.ndfa())
>>> db=NDFA2DFA(b.ndfa())
>>> da.witnessDiff(db)
@
```

The two regular expressions are not equivalent: the first one describes a language that contains the empty string but not the second one.

In the last example we could have defined a method that given two (or more) regular expressions would determine if they were equivalent. The possibility of constructing new methods from those defined in the package is one of the advantages of **FAdo**. Students can use the **FAdo** directly for training. For automatic assessment, teachers can use a Web system, based in simple CGIs or integrated in a Web learning environment, like Ganesh [10].

4. INTERACTIVE VISUALISATION

Currently the **FAdo** graphical environment allows the editing and visualisation of diagrams representing finite automata and provides an user interface to some conversion algorithms and string recognition. A diagram can be constructed from a finite automata definition, or created (or transformed) using the edit toolbox (at the right side of the interface). Figure 2 shows a diagram from the minimal `DFA` of Example 2 and its execution with string "`11012`". But more work must be done, improving the visualisation of the diagrams and the animation of the algorithms.

5. RELATED WORK

There are several projects whose goal is the manipulation of automata theory objects. Until 2000 few projects included a graphical interface. That was the case of the `Grail+` project [13]. `Grail+` operations are accessible either as individual programs (used as shell filters) or through a `C++`

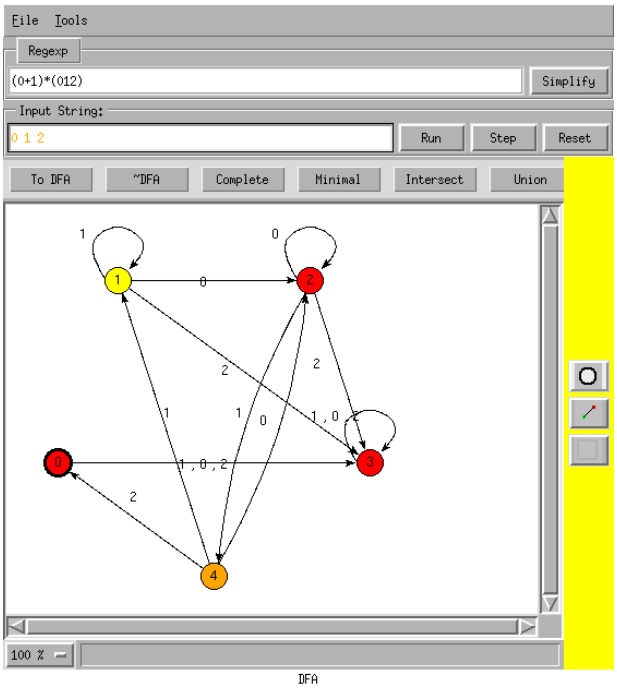


Figure 2: FAdo graphical interface

class library. In the `Grail`'s homepage [2], can be found a set of links to other automata theory software which includes `AUTOMATE` [5], `AMoRE` [9], `Fire Lite` [15], etc. In the last few years several `Java applets` for finite automata processing became available on-line, but normally their functionalities are very limited. `JFLAP` is one of the projects that shares most of our goals. Its main motivation is teaching and the current emphasis is in animation and interactive visualisation of automata theory concepts [8, 14]. It supports, in particular, drawing and execution of finite-state machines, conversions between deterministic and non-deterministic, and animated conversions from and to regular expressions. Compared with `JFLAP` one of the advantages of **FAdo** is its modular open source API and the possibility of constructing new methods. `Gaminal` is a project for the development of learning software for compiler design [6]. One of its components is an environment for processing finite automata. In particular, an electronic book that illustrates many of the regular languages manipulations, is available online [1]. Some interesting considerations about algorithms animations are found in [12].

6. FUTURE WORK

There are several different lines of future work. Because of space limitations we just highlight some of them. Algorithm animation is easily achieved by a step by step execution. However, this is not enough as a tool for helping understanding algorithms. We plan to obtain formal descriptions of the main concepts that are essential for a proof or an algorithm and to implement a specification language for a correct interactive manipulation. Besides extending the current API, theoretical topics as regular expressions simplification are being addressed. An integrated Web environment for publishing exercises and automatic assessment will be also available, building on previous experience of the members of the project team.

7. ACKNOWLEDGEMENTS

We thank the referees for their suggestions that helped to improve this paper.

8. REFERENCES

[1] Ganifa, generating finite automata. `http://rw4.cs.uni-sb.de/~ganimal/GANIFA/`,2004.

[2] Links to finite-state machines software. `http://www.csd.uwo.ca/research/grail/`,2004.

[3] GraphML file format. http://graphml.graphdrawing.org/, 2002.

[4] GraphViz. http://www.graphviz.org/, 2004.

[5] J. M. Champarnaud and G. Hanset. AUTOMATE, a computing package for automata and finite semigroups. *Journal of Symbolic Computation*, 12:197–220, 1991.

[6] S. Diehl and T. Kunze. Visualizing principles of abstract machines by generating interactive animations. *Future Generation Computer Systems*, 16(7):831–839, 2000.

[7] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 2nd edition, 2000.

[8] T. Hung and S. H. Rodger. Increasing visualization and interaction in the automata theory course. *SIGCSEB: SIGCSE Bulletin*, 32, 2000.

[9] V. Jansen, A. Potthoff, W. Thomas, and U. Wermuth. A short guide to the AMoRE system. Aachener informatik-berichte (90) 02, Lehrstuhl fur Informatik II, Universitat Aachen, January 1990.

[10] J. P. Leal and N. Moreira. Using matching for automatic assessment in computer science learning environments. In F. Restivo and L. Ribeiro, editors, *Web-Based Lerning Environments*, June 2000.

[11] M. Lutz. *Programming Python*. O'Reilly, 1996.

[12] H. W. P. M. Ayala-Rincón, A. da Fonseca and J. de Siqueira. A framework to visualize equivalences between computational models of regular languages. *Information Processing Letters*, 84(1):5–16, 2002.

[13] D. Raymond and D. Wood. Grail: A C++ library for automata and expressions. *J.Symbolic Computation*, 11, 1995.

[14] T. F. Ryan Cavalcante and S. H. Rodger. A visual and interactive automata theory course with jflap 4.0. In *Thirty-fifth SIGCSE Technical Symposium on Computer Science Education*, pages 140–144, 2004.

[15] B. W. Watson. The FIRE Lite: FAs and REs in C++. In *Proceedings of the First Workshop on Implementing Automata*, pages 167–188, 1996.

[16] S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1. Springer Verlag, 1997.

APPENDIX

A. PACKAGE DESCRIPTION

Constants	Printable	Description
Epsilon	@	empty string and regular expression ϵ
EmptySet	{}	empty set and regular expression $\emptyset$

Table 1: Constants

Class regexp

Class disj, inherits from regexp

Class star, inherits from regexp

Class concat, inherits from regexp

Method/Attribute	Description
__cmp__()	verifies if the two regexp are equivalent
ndfa()	NDFA equivalent to regexp
type()	returns the regexp type or value
empty()	tests if it is $\emptyset$
epsilon()	tests if it is ϵ
simplify()	applies simplification rules

Table 2: Classes for regular expressions

Class FA

Method/Attribute	Description
States	a list of states,
Sigma	a set of symbols
Initial	the initial state
Final	the set of final states
delta	a nested dictionary that associates a state to a transition
addstate()	adds a final state
validatestate()	checks if a state pertains to a FA
setinitial()	sets the initial state
setfinal()	sets a list of final states
setsigma()	defines the alphabet
addsigma()	adds a new symbol to the alphabet
completep()	checks if it is a complete FA
complete()	transforms in a complete FA
compact()	eliminates unused states
__len__()	returns the number of states

Table 3: Class for finite automata

Class NDFA, inherits from FA

Method/Attribute	Description
addTransition()	adds a new transition
EpsilonList()	ϵ-closure of a state
evalWord()	tests if the NDFA recognises a word
evalSymbol()	the set of the next possible states consuming a symbol
regexp()	regexp for current NDFA

Table 4: Class for nondeterministic finite automata

Class DFA, inherits from FA

Method/Attribute	Description
addTransition()	adds a new transition
evalWord()	tests if the DFA recognises a word
evalSymbol()	the set of the next possible states consuming a symbol
Minimal()	minimises the DFA
__cmp__()	verifies if the two automata are equivalent
uniqueStr()	unique string that gives us a DFA canonical form
__invert__()	DFA that recognises the complementary language
__or__()	DFA that recognises the union of two languages
__and__()	DFA that recognises the intersection of two languages
reverse()	DFA that recognises the reversal language
witness()	generates a word recognisable by the automata; or raises an exception
witnessDiff	witness for the difference of two DFA's or an exception is raised if theiy are equivalent
regexp()	regexp for current DFA

Table 5: Class for deterministic finite automata

Method/Attribute	Description
NDFA2DFA()	Returns a DFA equivalent to a NDFA given as argument. The method proceeds by subset construction.
isFA()	tests if the argument is a FA
isDFA()	tests if the argument is a DFA
isNDFA()	tests if the argument is a NDFA
readFromFile()	reads finite automata definitions from a file and returns a list of DFA's and/or NDFA's
saveToFile()	saves a finite automaton definition to a file
str2regexp()	parses a string and returns a regexp

Table 6: Module methods