FAdo: Interactive Tools for Learning Formal Computational Models*

Rogério Reis
Nelma Moreira
DCC-FC& LIACC, Universidade do Porto
R. do Campo Alegre 823, 4150 Porto, Portugal
{rvr,nam}@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. In particular we focus in the description of an interactive environment for editing and visualising finite automata and the conversion between automata and regular expressions.

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 math courses 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 graphically visualised. 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.

*Work partially funded by Fundação para a Ciência e Tecnologia (FCT) and Program POSI.
[1]The project page is `http://www.ncc.up.pt/fado`.

In this paper, we describe a collection of tools implemented in Python [2] 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. Section 3 introduces a graphical environment and some interactive visualisations. Ongoing work is summarised in Section 4.

2 Manipulating Regular languages

We assume basic knowledge of formal languages and automata theory [1]. An alphabet Σ is a nonempty set of symbols. A string over Σ is a finite sequence of symbols of Σ. The empty string is denoted by ϵ. The set $\Sigma^\star$ is the set of all strings over Σ. A language L is a subset of $\Sigma^\star$. If $L_1, L_2 \subseteq \Sigma^\star$, $L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$ and L_1^n is defined by $L_1^0 = \{\epsilon\}$, $L_1^n = L_1L_1^{n-1}$, for $n \geq 1$. The Kleene closure of a language L is defined by $L^\star = \bigcup_{n\in\mathbb{N}} L^n$. The set of regular languages over an alphabet Σ contains $\emptyset$, $\{\epsilon\}$, $\{a\}$ 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.

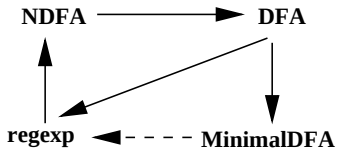

Figure 1: Conversions between regular language representations

In the next subsections we briefly describe how these representations are implemented as `Python` classes and which manipulations are currently available. A more detailed description can be found in [3].

2.1 Finite Automata

Formally a deterministic finite automaton (`DFA`) is specified by a 5-tuple $(S, \Sigma, \delta, s_0, F)$, where S is the set of states, Σ is the input alphabet,i.e. is a nonempty set of symbols, 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. If the transition function is total the `DFA` is said to be *complete*. In a nondeterministic automata (`NDFA`) 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 *language* accepted by a `DFA` $\mathcal{A}$ is $L(\mathcal{A}) = \{x \in \Sigma^\star \mid \delta(q_0, x) \in F\}$. The *language* accepted by `NDFA` $\mathcal{A}$ is $L(\mathcal{A}) = \{x \in \Sigma^\star \mid \delta(q_0, x) \cap F \neq \emptyset\}$. Two `FA`'s are *equivalent* if they accept the same language.

The class `FA` implements the basic structure of finite automata shared by deterministic and non-deterministic ones. This class also provides methods for manipulating this structures: add, set, delete, test, etc.

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`.

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 [1]. 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* [4]. 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 complementation, union and intersection.

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* [1]. 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()`.

2.2 Regular Expressions

A regular expression (r.e.) α over Σ represents a language $L(\alpha) \subseteq \Sigma^\star$ and is inductively defined by: $\emptyset$, ϵ and $a \in \Sigma$ are a r.e., where $L(\emptyset) = \emptyset$, $L(\epsilon) = \{\epsilon\}$ and $L(a) = \{a\}$; if α_1 and α_2 are r.e., $(\alpha_1 + \alpha_2)$, $(\alpha_1\alpha_2)$ and $\alpha_1^\star$ are r.e., respectively with $L((\alpha_1 + \alpha_2)) = L(\alpha_1) \cup L(\alpha_2)$, $L((\alpha_1\alpha_2)) = L(\alpha_1)L(\alpha_2)$ and $L({\alpha_1}^\star) = L(\alpha_1)^\star$. 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.

2.3 Converting Finite Automata to Regular Expressions

The standard conversion from `DFA`'s to regular expressions, we call it the *recursive* method, and 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 [1]. 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 [1]. 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 [1]. 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.

3 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).

The editing operations available are:

- add/move a state

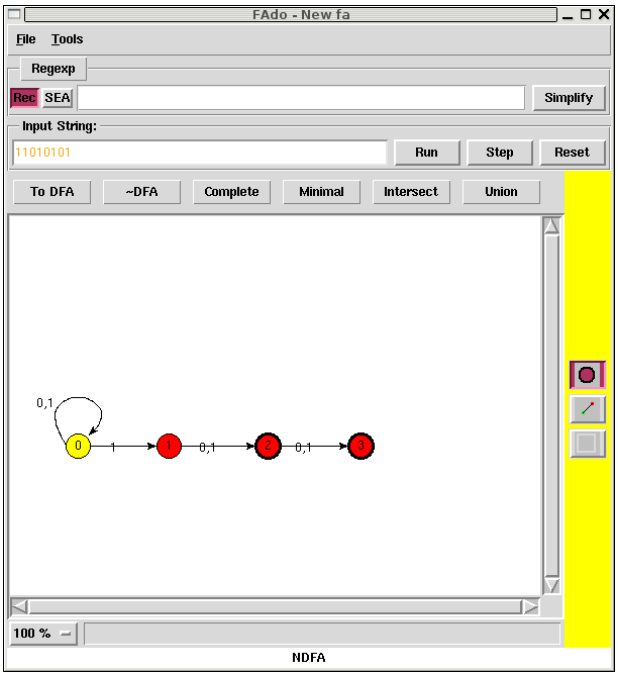

Figure 2: Creating a `NDFA` with the graphical interface.

- add a transition between two states (or a loop in one state); a label is prompted to the user.
- delete a state or a transition

A state can be made initial (*yellow* (light grey) background) and/or final (thicker border), by *Clicking* `Button3` in it.

Figure 2 shows a diagram of a `NDFA` that accepts the strings over $\{0, 1\}$ that have a `1` in the second or third symbol from the right, for instance `11010101` or `00110011`, but not `101000`.

Given a `NDFA` or a `DFA`, an input string can be entered and evaluated. In Figure 3 the `NDFA` is recognising the string `11010101` and after processing the prefix `1101`. As the automaton is non-deterministic there can be several current states, in this case, states `0`, `2` and `3`, that are coloured *orange* (lighter grey).

In the current version, the graphical user interface provides access to some of the conversions presented in Section 2 and whose results are visualised. In particular we have:

`NDFA`→ `DFA` Pressing the `To DFA` button, the current automaton is transformed in a `DFA` (by the subset construction). In Figure 4 is presented a `DFA` obtained from the `NDFA` in Figure 2.

`DFA`→ `MinDFA` Pressing the `Minimal` button, the current `DFA` is converted into the minimal `DFA`. In Figure 5 is presented the minimal `DFA` equivalent to the `NDFA` in Figure 2.

Figure 3: An NDFA recognising the string 110101.

DFA→ regexp Pressing the Regexp button, a regular expression equivalent to the current DFA is obtained. Currently we have implemented two algorithms for this transformation: the recursive (Rec) and by state elimination (SEA). Neither of them give small regular expressions and some simplification can be obtained by pressing the Simplify button.

regexp→ NDFA Pressing return-key after inputing a regular expression, the diagram of an equivalent ϵ-NDFA is drawn.

DFA→ **complete** DFA Pressing the Complete button, the current DFA is transformed into a complete DFA.

DFA→ ¬DFA Pressing the ~DFA button the current DFA is converted into a DFA for the complementary language. For that, the DFA must be complete (then the transformation consists in exchanging the final states).

Intersection Using the Intersect button it is possible to obtain an automaton that is the intersection of several DFA's.

Union In the same way, pressing the Union button it is possible to obtain an automaton that is the union of several DFA's.

Comparison of two regexp To check if two regular expressions are equivalent we can choose the option Compare... in the Tools menu. If there is a regular expression in the interface that expression is given as default for the comparison. In Figure 6

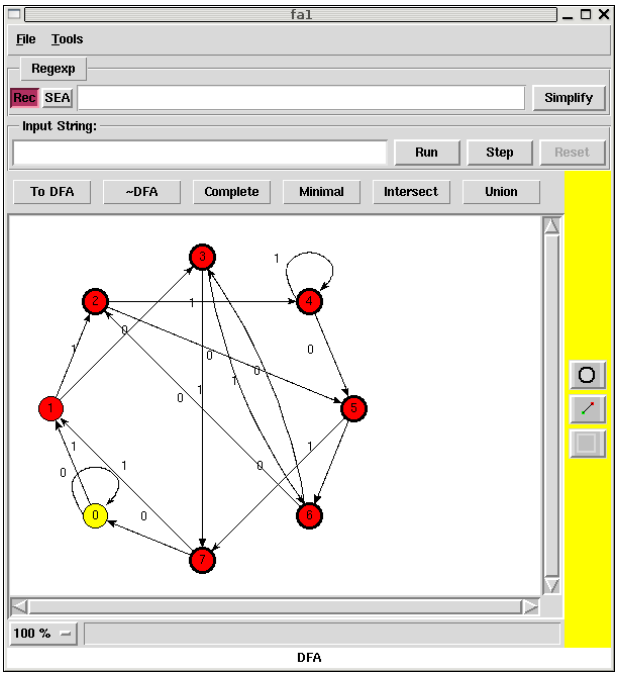

Figure 4: Conversion to `DFA`

is shown the dialog box for the comparison of two regular expressions: one obtained by converting the `NDFA` given in Figure 2 and the other a regular expression that can be obtained directly by the definition of the language. If the two regular expressions are not equivalent, a witness is returned, i.e., a string that is represented by the first but not by the second regular expression (as refered in Section 2.1.2).

4 Future Work

Much more work must be done, improving the visualisation of the diagrams and the animation of the algorithms. Several automata should be visualised at the same time. Better visualisation algorithms for automata diagrams must be designed and implemented. 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. An integrated Web environment for publishing exercises and automatic assessment will be also available.

References

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

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

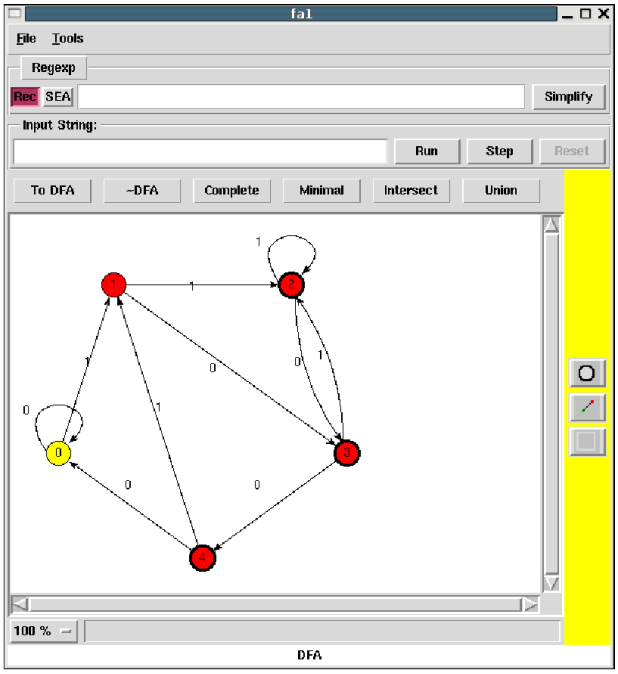

Figure 5: The minimal `DFA`

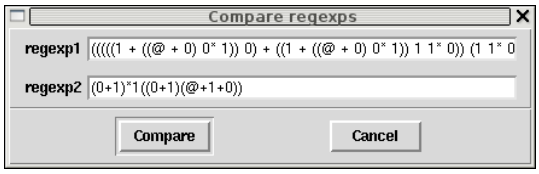

Figure 6: Comparing two regular expressions.

[3] Nelma Moreira and Rogério Reis. Interactive manipulation of regular objects with FAdo. In *Proceedings of 2005 Innovation and Technology in Computer Science Education (ITiCSE 2005)*. ACM, 2005.

[4] Rogério Reis and Nelma Moreira and Marco Almeida. On the Representation of Finite Automata In *Proceedings of the 7th Int. Workshop on Descriptional Complexity of Formal Systems (DCFS05)*, Como, Italy, June 30 - July 2. 2005.

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