

Davide Pereira Nabais

# DesCo: a Web-base Knowledge System for Descriptive Complexity of Formal Languages



Departamento de Ciência de Computadores Faculdade de  
Ciências  
da Universidade do Porto  
Junho / 2013

Davide Pereira Nabais

# DesCo: a Web-base Knowledge System for Descriptive Complexity of Formal Languages



*Tese submetida à Faculdade de Ciências da Universidade do  
Porto para obtenção do grau de Mestre em Ciência de Computadores*

Departamento de Ciência de Computadores Faculdade de Ciências  
da Universidade do Porto

Junho / 2013

# Resumo

A complexidade descritiva de linguagens formais estuda as medidas de descrições de linguagens em termos dos modelos de computação que as aceitam. Por exemplo, a *state complexity* de uma linguagem regular  $L$  é o menor número de estados necessários para um autômato finito determinístico aceitar  $L$ .

A investigação no domínio da complexidade descritiva nos últimos anos deu origem a uma grande colecção de resultados dispersos ao longo de algumas centenas de artigos. É cada vez mais difícil ter uma ideia geral de todo o trabalho feito nesta área de investigação .

DESCO, um sistema de conhecimento baseado na web para resultados de complexidade descritiva, foi criado para compensar esta dificuldade, bem como, para fornecer algumas funcionalidades que podem auxiliar os investigadores da área, enquanto tentam encontrar novos resultados.

Nesta tese, descrevemos os principais conceitos relacionados com complexidade descritiva, e como podem ser representados, de modo a que possam ser manipulados de forma a auxiliar a comunidade de investigação a encontrar novos resultados. Descrevemos também as ferramentas utilizadas na criação da base de conhecimento e interface web, e as funcionalidades do sistema.

# Abstract

Descriptive complexity of formal languages studies the measures of descriptions of languages in terms of their accepting models of computation. For example, the *state complexity* of a regular language  $L$  is the minimal number of states in any deterministic finite automaton accepting  $L$ .

The proliferous research in the field of descriptive complexity in recent years has given origin to a large collection of results dispersed along a few hundred articles. It is becoming increasingly difficult to have a general idea of all the work done in this field of research.

DESCO, a web-based knowledge system for descriptive complexity results, was created to compensate for this, as well as, to provide some features that could be of help to the researchers, when looking for new results.

In this thesis we describe the key concepts involved with descriptive complexity, and how they can be represented in some way so that they can be manipulated in order to aid the research community in finding new results. We also describe the tools used in the creation of the knowledge base and web interface, and the functionality of the system.

**À minha família**

# Acknowledgements

I thank Rogério Reis and Nelma Moreira for introducing me the world of scientific research, without which I would probably never have gained the experience I believe has made me better prepared to face future challenges, whether I work in research or the industry. I would also like to thank them for their guidance, patience and encouragement through the time we have worked together.

I thank Janusz Brzozowski for his criticism and contributions.

I am thankful to the research project CANTE (PTDC/EIA-CCO/101904/2008).

I would also like to thank my friends and family for their continued support, which has been of great help.

# Contents

<b>Resumo</b>	<b>3</b>
<b>Abstract</b>	<b>4</b>
<b>Acknowledgements</b>	<b>6</b>
<b>List of Figures</b>	<b>11</b>
<b>1 Introduction</b>	<b>12</b>
1.1 Thesis Outline . . . . .	14
1.2 Related Systems . . . . .	15
1.2.1 The Encyclopedia of Combinatorial Structures . . . . .	15
1.2.2 Dynamic Dictionary of Mathematical Functions . . . . .	16
1.2.3 The Navigator on Description Logic Complexity . . . . .	16
1.2.4 Minicomplexity . . . . .	16
1.2.5 OEIS . . . . .	17

<b>2</b>	<b>Basic Concepts</b>	<b>18</b>
2.1	Formal Languages . . . . .	18
2.2	Operations Over Languages . . . . .	19
2.3	Regular Languages . . . . .	20
2.3.1	Deterministic Finite Automata . . . . .	21
2.3.2	Non-Deterministic Finite Automata . . . . .	23
2.3.3	Regular Expressions . . . . .	24
2.4	Descriptive Complexity . . . . .	25
2.5	Conclusion . . . . .	28
<b>3</b>	<b>The Knowledge Base</b>	<b>29</b>
3.1	Language Classes . . . . .	29
3.2	Models of Computation . . . . .	30
3.3	Complexity Measures . . . . .	31
3.4	Operations . . . . .	31
3.5	Operational Complexities . . . . .	32
3.6	Language Families . . . . .	35
3.6.1	The Universal Witness . . . . .	38
3.7	Conclusion . . . . .	40
<b>4</b>	<b>Desco Components</b>	<b>42</b>



4.1	Web Framework . . . . .	42
4.2	Database Interaction . . . . .	43
4.3	Mathematical Typesetting . . . . .	45
4.4	Formal Language Symbolic Manipulator . . . . .	46
4.5	Conclusion . . . . .	48
<b>5</b>	<b>The Web Interface</b>	<b>49</b>
5.1	Introducing and Updating Information . . . . .	49
5.2	Consulting the Knowledge Base . . . . .	51
5.3	Interacting with FAdo . . . . .	53
5.4	Conclusion . . . . .	54
<b>6</b>	<b>Conclusion</b>	<b>55</b>
6.1	Current State of DesCo . . . . .	55
6.2	Future Work . . . . .	56
<b>A</b>	<b>Web Interface Screenshots</b>	<b>58</b>
	<b>References</b>	<b>61</b>

# List of Figures

2.1	DFA $\mathcal{D}$ . . . . .	23
2.2	DFA $A$ (left) and DFA $B$ (right) recognize the language $L$ . . . . .	26
2.3	NFA $C$ (left) and its equivalent minimal DFA (right) . . . . .	26
2.4	Language Family $\mathcal{A}_n$ . . . . .	27
3.1	A family of minimal DFAs recognizing $L_m$ . . . . .	35
3.2	DFA $\mathcal{A}$ . . . . .	36
3.3	$\mathcal{U}_n(a, b, c)$ . . . . .	40
4.1	Web browser displaying a MathJax generated formula . . . . .	46
5.1	Form for adding a new operation . . . . .	50
5.2	Listing operations . . . . .	51
5.3	Language family details . . . . .	52
5.4	Listing results for several operations and language classes . . . . .	53
A.1	Listing results for union operation . . . . .	58

A.2	Details for a particular result . . . . .	59
A.3	FAdo generating instances . . . . .	59
A.4	FAdo operating with universal witnesses . . . . .	60

# Chapter 1

## Introduction

Recently, the descriptive complexity of formal languages has been extensively researched [GKK<sup>+</sup>02, Hro02, Yu05, HK11, Brz10]. Descriptive complexity of formal languages studies the measures of descriptions of languages in terms of their accepting models of computation. For example, the *state complexity* of a regular language  $L$  is the minimal number of states in any deterministic finite automaton accepting  $L$ . For each measure, it is important to know the size of the smallest representation for a given language as well as how the size varies when several such representations are combined or transformed. These studies are of much interest, as they relate to the efficiency of implementing operations on languages. For instance, given a binary operation  $\circ$  closed for regular languages, then the study in state complexity looks to express the worst-case state complexity of  $L_1 \circ L_2$  as a function of the state complexities of  $L_1$  and  $L_2$ . This is crucial in new applied areas where automata and other models of computation are used, for instance, for pattern matching in bioinformatics or network security, or for model checking or security certificates in formal verification systems. In general, having succinct objects will improve our control on software, which may become smaller, more efficient and easier to certify.

Among formal languages, regular languages are fundamental structures in computer science. Despite their apparently weak expressive power (lowest level of Chomsky hierarchy), regular languages have applications in almost all areas of computer science. The general decidability of their properties and operations, and in many cases the linear or low polynomial computational complexity of the available algorithms, is also an attractive property for this class of languages. In particular, when compared with the undecidability world of context-free languages, just in the next level of Chomsky hierarchy. Therefore, it is essential that the structural properties of regular language representations are researched deeper. One of the most studied complexity measures for regular languages is the state complexity. The state complexity of an operation over languages is the complexity of the resulting language as a function of the complexities of its arguments. Both concepts can be extended to other models of computation (e.g. nondeterministic automata, two-way automata, regular expressions, grammars, etc.), other measures (number of transitions, number of symbols, etc.) and other classes of languages (classes of sub-regular languages, context-free languages, recursive languages, etc.). Knowing the descriptive complexity and succinctness of the objects has also obvious consequences for the computational complexity of the algorithms that manipulate them.

This proliferous research gave origin to a multitude of results that are scattered over a few hundred articles, with the inevitable lack of unified terminology and notation. This makes it very difficult to have a global perspective of this field and realize what is the current coverage achieved in order to know where to allocate more research efforts. All these different aspects and the huge number of results obtained, mainly in the last couple of decades, motivates the need of a tool that helps to structurally organize, visualize and manipulate this information. In this way, researchers and software engineers working in applications based on automata and formal languages can also more easily have access to information that can help to improve the performance of

their algorithms. Moreover, such a tool could be of great help to anyone refereeing for a conference or journal of this field, by providing a way to quickly look up existing results, giving a higher degree of confidence to the referee when deciding whether to accept or not a submitted article.

This motivated the creation of the DESCo system, a Web-based knowledge system for results in descriptonal complexity. This is not a trivial task as most of the concepts are abstract and difficult to classify and instantiate. For instance, which are the main concepts to describe the complexity of a language operation; which is the best way to represent parameterized families of languages and how to determine if two different representations correspond to the same family; etc...

## 1.1 Thesis Outline

The thesis is structured as follows:

- Section 1.2 gives an overview of related systems, in the sense that they deal with the same kind of data.
- Chapter 2 gives the reader an introduction to some concepts of formal languages, mostly concentrating on regular languages, and a few examples of descriptonal complexity.
- Chapter 3 describes how the information is represented in the knowledge base, while introducing some additional concepts not covered in Chapter 2.
- Chapter 4 gives a brief overview of the tools utilized to implement the DESCo system, and some small examples of how they can be put to use.

- Chapter 5 describes the structure and functionality of the web interface. It discusses introducing, updating and consulting the knowledge base through the web interface, and how the user can interact with a formal language symbolic manipulator.
- Chapter 6 has some final remarks regarding this thesis, the current state of DESCO and possible future work.

## 1.2 Related Systems

There are some similar systems that deal with (somehow) similar data but with different aims and solutions. Neil Sloane's *The On-Line Encyclopedia of Integer Sequences* [The11] collects about 200,000 sequences of numbers the first collection of which was published as a book in 1970's [SP95]. A smaller and more recent project is the *The Encyclopedia of Combinatorial Structures* [Alg11b]. The same Web site includes a *Dictionary of Mathematical Functions* [Alg11a]. The *Complexity Zoo* [Aar11] is a Wiki that contains information about computational complexity classes and related topics. More sophisticated but also more restricted is *The Navigator on Description Logic Complexity* [Zol11] where results on the computational complexity of reasoning in Description Logics can be browsed. More recently, *mini-complexity* [Kap12] website was created, which focuses on the complexity of two-way finite automata. These systems are briefly described in the following subsections.

### 1.2.1 The Encyclopedia of Combinatorial Structures

*The Encyclopedia of Combinatorial Structures* was created by Stéphanie Petit, and stores integer sequences along with their associated decomposable combinatorial struc-

tures, making it possible to automatically compute several properties such as generating functions, closed formulas, asymptotic estimates, etc.

### 1.2.2 Dynamic Dictionary of Mathematical Functions

The *Dynamic Dictionary of Mathematical Functions* is a website consisting of interactive tables of mathematical formulas on elementary and special functions. The website can interactively provide differential equations, plot functions, numerical evaluations, Taylor expansions, among other features.

### 1.2.3 The Navigator on Description Logic Complexity

The *Description Logic Complexity Navigator* is a web page that allows to review the complexity of reasoning tasks of various description logics by adding or removing features to a logic. It includes a comprehensive list of references to the literature. It is currently maintained by Evgeny Zolin.

### 1.2.4 Minicomplexity

Minicomplexity theory is a mathematical theory that studies the complexity of two-way finite automata. It focuses on the resource requirements for solving certain problems.

The *minicomplexity* website, features a list of several machines, and their description, commonly used in minicomplexity theory, related complexity classes and usual problems, such as acceptance and inclusion. The project is currently under development, and does not yet feature reductions between problems. It also features an extensive



bibliography.

### 1.2.5 OEIS

The *On-Line Encyclopedia of Integer Sequences*(OEIS), is an online database of integer sequences, created by N. J. A. Sloane, and is currently maintained by *The OEIS Foundation Inc.* The OEIS records information on integer sequences of interest to mathematicians in general, and is widely cited. It contains over 200,000 sequences, very likely making it the largest database of its kind. Each entry lists the initial terms (if available), a description, formulae, programs to generate the sequence, references, links to relevant web pages, and more, including the option to generate a graph or play a musical representation of the sequence.

Searching the OEIS can be done by entering a few terms of a sequence, by entering a string such as “Pascals triangle” or “Fibonnaci sequence”, or by entering a sequence ID number.

The OEIS can have many applications. The most obvious way to use it, would be to look up a sequence, very much like looking up a word in a dictionary. If one encounters a sequence during some calculations, and want to find out if there is a known formula for the  $n$ -th term, the OEIS can be of great use. Another application would be to find out the latest information about certain problems, such as, the decimal expansion of  $\pi$ . The OEIS can also be used to simplify complicated expressions. By calculating the first few terms of a complicated expression, and looking up the sequence, one might find out there is known simplified version of our expression. The OEIS can also be of use when trying to figure out if two different objects can be enumerated by the same sequence, since by establishing bijections between objects, it is possible to conclude new properties from this relation.

# Chapter 2

## Basic Concepts

This chapter gives the reader an introduction to some concepts of formal languages. These concepts are in accordance to Ullman and Hopcroft's *Introduction to Automata Theory* [HMU06], which can be consulted for more details. We will concentrate on regular languages, as most of the data currently available in DESCO is related with the descriptive complexity of regular and sub-regular languages.

### 2.1 Formal Languages

An *alphabet*, is a finite set of symbols, usually denoted by  $\Sigma$ . A *word* is a sequence of symbols from an alphabet. A *language* is a set of words over a given alphabet. The set of all words over an alphabet  $\Sigma$  is denoted by  $\Sigma^*$ . The length of a word  $w$  over an alphabet  $\Sigma$ , is the number of occurrences of symbols from  $\Sigma$  in  $w$ , and it is denoted by  $|w|$ . The empty word, i.e.,  $|w| = 0$ , is denoted by  $\epsilon$ . For a given  $k \geq 0$ ,  $\Sigma^k$  is the set of all words over  $\Sigma$  of length  $k$ . A *language class* is a set of languages which can be represented by a particular model of computation or have specific properties

in common. A *model of computation* is a finite representation of a language.

## 2.2 Operations Over Languages

Since languages are sets of words, all boolean operations usually defined over sets are also defined over languages. Let  $L_1$  and  $L_2$  be two languages over some alphabet  $\Sigma$ . The definition of *union*, *intersection*, *symmetric difference*, *difference* and *complement* follows:

- Union:  $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ or } w \in L_2\}$
- Intersection:  $L_1 \cap L_2 = \{w \mid w \in L_1 \text{ and } w \in L_2\}$
- Symmetric Difference:  $L_1 \oplus L_2 = (L_1 \cup L_2) \setminus (L_1 \cap L_2)$
- Difference:  $L_1 \setminus L_2 = \{x \in \Sigma^* \mid x \in L_1 \text{ and } x \notin L_2\}$
- Complement:  $\overline{L_1} = \Sigma^* \setminus L_1$

Many other operations over languages are defined. A few examples are *reversal*, *(con)catenation*, *Kleene closure* and *left quotient*. For a language  $L$  over an alphabet  $\Sigma$ , the reversal of  $L$  is denoted by  $L^R$ , and is defined by  $L^R = \{w^R \mid w \in L\}$ , where  $w^R = a_{n-1} \cdots a_0$ , if  $w = a_0 \cdots a_{n-1}$  and  $\forall_i a_i \in \Sigma$ . The concatenation of two languages  $L_1$  and  $L_2$  is denoted by  $L_1 L_2$  and is defined by  $L_1 L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$ . The Kleene closure (also know as Kleene operator or Kleene star) of a language  $L$  can be defined as  $L^* = \{x_1 x_2 \dots x_n \mid n \geq 0 \text{ and } x_i \in L, 1 \leq i \leq n\}$ . The left quotient of  $L_2$  by  $L_1$ , denoted by  $L_1^{-1} L_2$ , is the language  $\{y \in \Sigma^* \mid xy \in L_2 \text{ and } x \in L_1\}$ . If  $|L_1| = 1$ , usually the operation is called *left quotient by a word* and is denoted by  $w^{-1} L = \{x \in \Sigma^* \mid wx \in L\}$ . An infinite number of operations can be defined

if one takes into consideration combined operations, for example  $(L_1 \cup L_2)^*$ ,  $L_1 L_2^R$ ,  $L_1((L_2 \cup L_3)^*)^R$  and so on.

## 2.3 Regular Languages

The Chomsky hierarchy is a containment hierarchy language classes. The hierarchy consists of the following classes: *Recursively Enumerable*, *Context-Sensitive*, *Context-Free* and *Regular*. Regular languages are the most restricted, and the simplest, languages in the Chomsky hierarchy.

The set of regular languages over an alphabet  $\Sigma$  can be defined recursively as follows:

- The empty language  $\emptyset$  is regular
- The empty word language  $\{\varepsilon\}$  is regular
- For each  $a \in \Sigma$ , the language  $\{a\}$  is regular
- If  $A$  and  $B$  are both regular languages, then  $A \cup B$ ,  $AB$  and  $A^*$  are regular languages
- No other languages over  $\Sigma$  are regular

Models of computation that recognize exactly the set of regular languages include deterministic finite automata (DFA), non-deterministic finite automata (NFA) and regular expressions (RE). Two important subclasses of regular languages are *finite languages* and *star-free languages*.

Finite languages are those containing only a finite number of words. Finite languages are regular, as they can be defined as the union of singletons associated to each word

in the language. Star-free languages are those that can be described without making use of the Kleene star operation. Star-free languages include all finite languages.

### 2.3.1 Deterministic Finite Automata

A deterministic finite automaton is a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$ , consisting of:

- A finite set of states  $Q$
- A finite set of symbols  $\Sigma$ , called the alphabet
- A transition function  $\delta : Q \times \Sigma \rightarrow Q$
- An initial (or starting) state  $q_0 \in Q$
- A set of accepting states  $F \subseteq Q$ , also known as set of final states

Given a DFA  $\mathcal{A}$  and a word  $w$  as input,  $\mathcal{A}$  accepts  $w$ , if  $w$  is contained in the language recognized by  $\mathcal{A}$ , denoted by  $\mathcal{L}(\mathcal{A})$ , and rejects  $w$  otherwise. To do this, we start applying the transition function  $\delta$  to the initial state  $q_0$  and the first symbol of our input word  $w = a_0a_1 \cdots a_{n-1}$ . After evaluating  $\delta(q_0, a_0) = q_i$ , we consume the next symbol  $a_1$ , evaluating  $\delta(q_i, a_1) = q_j$ , and continue to do so until we consume the last symbol of  $w$   $a_{n-1}$ , by evaluating  $\delta(q', a_{n-1}) = q''$ . Finally, if  $q'' \in F$  then  $w \in \mathcal{L}(\mathcal{A})$ , otherwise  $w \notin \mathcal{L}(\mathcal{A})$ .

An extended transition function  $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$  can be defined by induction on the size of the input word as follows:

- $\hat{\delta}(q, \varepsilon) = q$ , i.e., if we are in state  $q$  and consume no symbols, then we stay in state  $q$

- $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$

The language accepted by a DFA  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  is the set of all words that take the initial state to one of the final states, i.e.,  $\mathcal{L}(\mathcal{A}) = \{p \in \Sigma^* \mid \hat{\delta}(q_0, p) \in F\}$ . It is possible to prove that a language is regular, by defining a DFA that recognizes it.

For each regular language, there exists a *minimal* DFA, i.e., a DFA with a minimum number of states. The *Nerode congruence* [Ner58]  $\approx_L$  of  $L \subseteq \Sigma^*$  is defined as follows:

$$\text{for } x, y \in \Sigma^*, x \approx_L y \text{ if and only if } xv \in L \Leftrightarrow yv \in L \text{ for all } v \in \Sigma^*$$

The relation  $\approx_L$  is an equivalence relation on strings, which divides the set of all finite strings into equivalence classes. Each equivalence class is called a quotient of the language. The language  $L$  is regular if and only if  $\approx_L$  has a finite number of equivalence classes. The number of states in the minimal DFA that recognizes  $L$  is equal to the number of equivalence classes in  $\approx_L$ . This implies that the minimal DFA is unique.

DFA are commonly represented as digraphs with labeled edges, where to each state we associate a node. For each state  $q \in Q$  and each symbol  $a \in \Sigma$ , if  $\delta(q, a) = q'$  then our digraph has an edge from node  $q$  to node  $q'$  with label  $a$ . The initial state  $q_0$  is marked with an arrow, and each of the final states is marked with a double circle. Fig. 2.1 shows the digraph associated to the DFA  $\mathcal{D} = (\{st_0, st_1, st_2, st_3, st_4, st_5\}, \{a, b, c\}, \delta, st_0, \{st_3, st_4\})$  where  $\delta$  is

$$\delta(st_0, a) = st_1 \quad \delta(st_0, b) = st_5 \quad \delta(st_0, c) = st_5$$

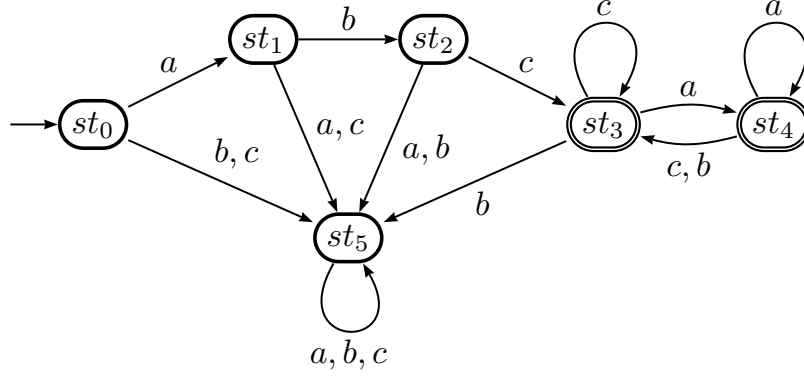
$$\delta(st_1, b) = st_2 \quad \delta(st_1, a) = st_5 \quad \delta(st_1, c) = st_5$$

$$\delta(st_2, c) = st_3 \quad \delta(st_2, a) = st_5 \quad \delta(st_2, b) = st_5$$

$$\delta(st_3, a) = st_4 \quad \delta(st_3, c) = st_3 \quad \delta(st_3, b) = st_5$$

$$\delta(st_4, a) = st_4 \quad \delta(st_4, b) = st_3 \quad \delta(st_4, c) = st_3$$

$$\delta(st_5, a) = st_5 \quad \delta(st_5, b) = st_5 \quad \delta(st_5, c) = st_5$$

Figure 2.1: DFA  $\mathcal{D}$ 

### 2.3.2 Non-Deterministic Finite Automata

To represent regular languages, it can be convenient to use a more flexible and compact representation. Using non-deterministic finite automata to represent regular languages, usually requires a smaller number of states and applying some operations to NFA can be much more straight forward when compared to DFA.

An NFA is a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$ , very similar to a DFA, consisting of:

- A finite set of states  $Q$
- A finite set of symbols  $\Sigma$
- A transition function  $\delta : Q \times \Sigma \rightarrow 2^Q$
- An initial (or starting) state  $q_0 \in Q$
- A set of final states  $F \subseteq Q$

where  $2^Q$  is the *power set* of  $Q$ , i.e., the set of all subsets of  $Q$ . Sometimes slight variations of this definition are considered, such as, multiple initial states and NFA with  $\varepsilon$  moves. These variations are considered as they can be handy regardless of not adding any expressiveness to the model of computation.

Variations aside, the only difference between DFA and NFA, is the transition function. In DFA,  $\delta$  maps a state and a symbol to a state. The NFAs transition function maps a state and a symbol to a set of states, meaning that the NFA can be in multiple states at once, hence the *non-determinism*. To verify if a word is accepted by an NFA, we process the input like when using DFA, but taking into consideration the multiple states the NFA can be in. When the input word is fully processed, we must check if at least one of the possible states the NFA can be in is a final state. If so, then the NFA accepts the word. If none of the states is final, then the NFA rejects the word.

More formally, given an NFA  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ , we can define an extended transition function  $\hat{\delta} : 2^Q \times \Sigma^* \rightarrow 2^Q$ , as:

- $\hat{\delta}(P, \varepsilon) = P$ , where  $P \subseteq Q$
- $\hat{\delta}(P, aw) = \hat{\delta}(\bigcup_{s \in P} \delta(s, a), w)$

The language recognized by  $\mathcal{A}$  can be defined by:

$$\mathcal{L}(\mathcal{A}) = \{p \in \Sigma^* \mid \hat{\delta}(\{q_0\}, p) \cap F \neq \emptyset\}$$

### 2.3.3 Regular Expressions

Sometimes an algebraic description of regular languages is more useful than automata, for example, in pattern matching applications. To this end, regular expressions are used as a declarative representation of the words we want to accept.



We can define RE recursively as follows:

- $\varepsilon$  and  $\emptyset$  are RE and,  $L(\varepsilon) = \{\varepsilon\}$  and  $L(\emptyset) = \emptyset$
- Let  $a \in \Sigma$ , then  $a$  is a RE and  $L(a) = \{a\}$
- If  $r_1$  and  $r_2$  are RE, then  $r_1 + r_2$  is also a RE, and  $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
- If  $r_1$  and  $r_2$  are RE, then  $r_1 r_2$  is also a RE, and  $L(r_1 r_2) = L(r_1) L(r_2)$
- If  $r$  is a RE, then  $r^*$  is also a RE, and  $L(r^*) = (L(r))^*$

As an illustrative example, we can consider the DFA  $\mathcal{D}$ , from Section 2.3.1, represented in Fig. 2.1

The language defined by  $\mathcal{D}$  can be represented by the RE  $abc(a + (ab)^* + c)^*$ , which is arguably a more readable and compact representation.

## 2.4 Descriptive Complexity

Two simple examples of descriptive complexity follow, to help the reader become more familiarized with the concept. Consider the following language, over the unary alphabet  $\Sigma = \{a\}$ ,  $L = \{a^{2n} \mid n \geq 0\}$  (or, in other words,  $L = \{\epsilon, aa, aaaa, aaaaaa, \dots\}$ , where  $\epsilon$  represents the empty-word). Since the language  $L$  is regular, it can be recognized by a deterministic finite automaton. In fact, it can be recognized by infinitely many DFAs. To illustrate this fact, two examples of DFAs that recognize  $L$  are given in Fig. 2.2. Since DFA  $B$  is the (state) minimal DFA that recognizes  $L$ , we say that the descriptive complexity, in respect to the number of states, i.e., the state complexity, of  $L$ , denoted by  $sc(L)$ , is 2.

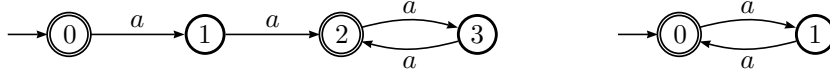


Figure 2.2: DFA  $A$  (left) and DFA  $B$  (right) recognize the language  $L$

Let us look at another example. In Fig. 2.3, we present a nondeterministic finite automaton  $C$  with three states. It is well known that any  $n$ -state NFA, can be converted, via *subset construction*, to an equivalent DFA, i.e. that recognize the same language, with at most  $2^n$  states. This conversion is called *determinization*. If we apply the *subset construction* to NFA  $C$ , we will obtain the DFA shown in Fig. 2.3, which is minimal, and thus we can conclude that the state complexity of the language recognized by NFA  $C$ , or  $sc(\mathcal{L}(C))$ , is  $2^3 = 8$ , whereas the non-deterministic state complexity of the same language,  $nsc(\mathcal{L}(C))$ , is 3, i.e., the number of states of a minimal NFA accepting  $\mathcal{L}(C)$

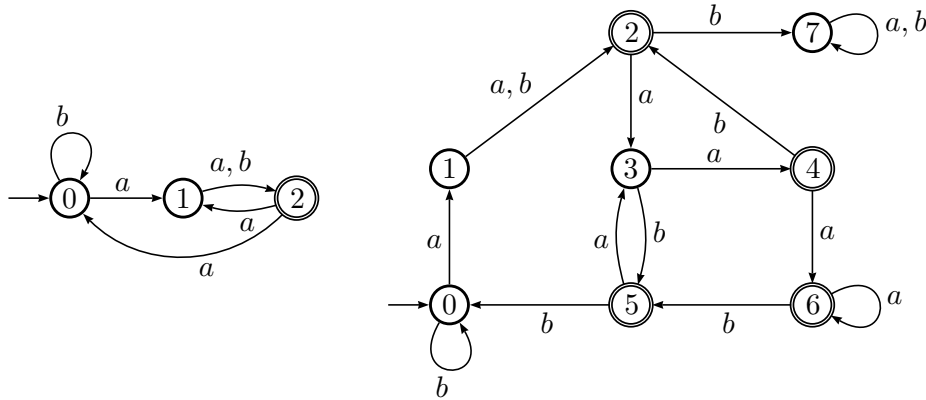


Figure 2.3: NFA  $C$  (left) and its equivalent minimal DFA (right)

We conclude this section by giving a more practical example of descriptive complexity, the state complexity of star operation on regular languages. This result was obtained by Yu *et al.* [YZS94] and states that for an  $n$ -state DFA  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ , such that  $|F - q_0| \geq 1$  and  $n > 1$ ,  $2^{n-1} + 2^{n-2}$  states are sufficient and necessary, in the worst case, for a DFA to accept  $\mathcal{L}(\mathcal{A})^*$ .

One can start by proving that for any  $n$ -state DFA  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ , such that  $|F - q_0| \geq 1$ ,  $2^{n-1} + 2^{n-2}$  states are sufficient to accept  $\mathcal{L}(\mathcal{A})^*$ , by providing a method to construct a new DFA  $\mathcal{A}'$ , such that  $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})^*$ , and that the number of states of  $\mathcal{A}'$  is bounded by  $2^{n-1} + 2^{n-2}$ . The construction of DFA  $\mathcal{A}' = (Q', \Sigma, \delta', q'_0, F')$  follows:

- $q'_0 \notin Q$  is the new initial state,
- $Q' = \{q'_0\} \cup \{P \mid P \subseteq (Q - F - \{q_0\}) \text{ and } P \neq \emptyset\}$   
 $\cup \{R \mid R \subseteq Q \text{ and } q_0 \in R \text{ and } R \cap (F - \{q_0\}) \neq \emptyset\}$ ,
- $\delta'(q'_0, a) = \{\delta(q_0, a), q_0\}$  if  $\delta(q_0, a) \in F - \{q_0\}$ ,  
and  $\delta'(q'_0, a) = \{\delta(q_0, a)\}$  otherwise.  
 $\delta'(R, a) = \delta(R, a)$  for  $R \subseteq Q$  and  $a \in \Sigma$  if  $\delta(R, a) \cap (F - \{q_0\}) = \emptyset$ ,  
and  $\delta'(R, a) = \delta(R, a) \cup \{q_0\}$  otherwise,
- $F' = \{q'_0\} \cup \{R \mid R \subseteq Q \text{ and } R \cap F \neq \emptyset\}$ .

The DFA  $\mathcal{A}'$  recognizes  $\mathcal{L}(\mathcal{A})^*$ , and  $|Q'| \leq 2^{n-1} + 2^{n-2}$ .

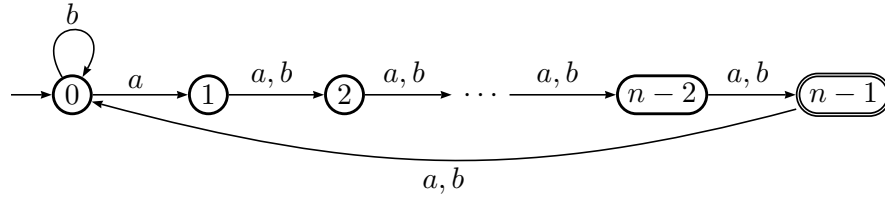


Figure 2.4: Language Family  $\mathcal{A}_n$

To prove that, in the worst case,  $2^{n-1} + 2^{n-2}$  states are necessary, we can provide a family of languages. For  $n = 2$ ,  $L_1 = \{w \in \{a, b\}^* \mid \#_a(w) \text{ is odd}\}$ , as it is accepted by a 2-state DFA, and  $L_1^* = \{\varepsilon\} \cup \{w \in \{a, b\}^* \mid \#_a(w) \geq 1\}$  is accepted by a DFA with 3 states. For  $n \geq 3$ , the family  $\mathcal{A}_n$  shown in Fig. 2.4 can be considered.

Constructing a DFA  $\mathcal{A}'_n$  from  $\mathcal{A}_n$  as previously described, it is possible to show that every state in  $\mathcal{A}'_n$  is reachable from the initial state, and that each state defines a distinct equivalence class, in terms of the Nerode congruence, proving that  $2^{n-1} + 2^{n-2}$  states are necessary, in the worst case, to accept the star of an  $n$ -state DFA.

## 2.5 Conclusion

In this chapter an introduction to some concepts of formal languages was given. We gave some elemental definitions and notation on languages, operations, automata, regular expressions and descriptonal complexity.

For deterministic finite automata, the formal definition and a description of how they can recognize a regular language was given. It was also mentioned the existence of a unique minimal DFA for each regular language.

It is important to note that many other models of computation and languages classes can be defined, as well as the descriptonal complexity of their operations.

# Chapter 3

## The Knowledge Base

In this chapter we describe how the information about descriptonal complexity is represented in the knowledge base. The most important concepts we are going to consider are: *Language Classes*, *Models of Computation*, *Language Operations*, *Language Families*, *Complexity Measures* and *Operational Complexities*. Along with the basic information, some additional details, that will be useful in the future, are also taken into consideration. For example, data for interacting with the FADO system, such as which FADO method generates a deterministic automaton for a given language family, or performs an operation between some given languages.

We briefly describe each of the above mentioned concepts.

### 3.1 Language Classes

A language class is defined with a name and a description. Given a preorder on language classes, a hierarchy can be considered. Currently, we define an inclusion hierarchy. In addition, witnesses of non-emptiness of language classes are available.

As already mentioned, the class of regular languages, for instance, can be defined recursively, over an alphabet  $\Sigma$ , as follows:

- The empty language  $\emptyset$  is regular
- The empty string language  $\{\varepsilon\}$  is regular
- For each  $a \in \Sigma$ , the language  $\{a\}$  is regular
- If  $A$  and  $B$  are both regular languages, then  $A \cup B$ ,  $A \cap B$ ,  $AB$  and  $A^*$  are regular languages
- No other languages over  $\Sigma$  are regular

## 3.2 Models of Computation

Models of computation are represented by a name, an abbreviation, a description and the FADO's class that corresponds to it. The description can store a wide variety of information, such as a mathematical description or certain properties that hold for a specific model.

A deterministic finite automaton, for example, would have DFA, as its abbreviation, `FAdo.fa.DFA` as its FADO class, and could be described as a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$  consisting of:

- a finite set of states  $Q$
- a finite set of input symbols  $\Sigma$ , called the alphabet
- a transition function  $\delta : Q \times \Sigma \rightarrow Q$ :
- a initial state  $q_0 \in Q$

- a set of final states  $F \subseteq Q$

### 3.3 Complexity Measures

A complexity measure of a language is a function of a size of an associated model of computation. Complexity measures are described by a name, a description and the associated model of computation. The name, for example, could be *state complexity* and its description, “*The state complexity of a regular language  $L$ , denoted by  $sc(L)$ , is the number of states of its minimal DFA*”.

Complexity measures are usually referenced, in descriptonal complexity results, by a variable, instead of their shorthand notation. For example, state complexity is typically referred to by  $n$  or  $m$ , instead of  $sc(L)$ . For this reason, we choose to include these commonly used variables when describing complexity measures.

### 3.4 Operations

Operations on formal languages include all boolean operations usually defined on sets plus other specific operations. For instance, concatenation of two languages  $L_1$  and  $L_2$ , is defined by  $L_1L_2 = \{w_1w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$ . We also consider the simulation (conversion) of different models of the same language. To characterize an operation we use a name, a symbol (represented in  $\text{\LaTeX}$ ), a description (as illustrated above), its arity, if it is a combined operation or not, and, in the case of being a combined operation, which operations it is composed of. The purpose of the  $\text{\LaTeX}$  symbol is mainly for displaying a more intuitive label than just the name of the operation, but also an attempt to standardize symbols used in the literature.

To represent combined operations, we chose to use a tree structure. Suppose we want to define the following operation:  $O = L_1 \cup L_2^*$  (union of Kleene star). The operation  $O$  can be defined considering that the first operation is  $\cup$  (union), the second one is  $*$  (Kleene star) and that it is applied to the second argument of the first operation. Every combined operation can be defined recursively in this manner.

### 3.5 Operational Complexities

The descriptive complexity of an operation over a class of languages, for a given measure and a given model, is the (worst-case) complexity of a language resulting from the operation, considered as a function of the descriptive complexity of its arguments. For instance the *state complexity of a binary operation  $\circ$  on regular languages* can be stated as the following decision problem:

- Given an  $m$ -state DFA  $A_1$  and an  $n$ -state DFA  $A_2$ .
- How many states are sufficient and necessary, in the worst case, to accept the language  $L(A_1) \circ L(A_2)$  by a DFA?

To obtain an upper bound, usually, an algorithm is provided, such that, given DFAs as the operands, constructs a DFA that accepts the resulting language. The number of states of this DFA (as a function of the state complexities of the operands) is an upper bound for the state complexity of the referred operation. To show that an upper bound is tight, a family of languages (one language, for each possible value of the state complexity) can be given, for each operand, such that the minimal automata resulting from the operation (i.e the state complexity of resulting language) achieve that bound, if not, then they at least provide a lower bound.



The same approach can be used to obtain other operational complexities. To specify an operational complexity, the following information is considered:

**Operation** the operation that we are considering;

**Complexity kind** if it is worst-case or average-case;

**Complexity measure** as defined above, e.g. *state complexity*;

**Argument types** which computational models are used and to which language classes the arguments must belong to;

**Result type** what model represents the resulting language;

**Complexity function** the operation complexity as function of the arguments complexity;

**An algorithm** that for a given model of computation calculates the correspondent model of the resulting language having as input models for the argument languages;

**Alphabet size** The operational complexity (and witnesses) can depend on the size of the languages alphabet;

**Tightness** whether the complexity function is reachable or not;

**Restrictions on argument parameters** For which values of the parameters the complexity function applies (e.g. is an upper bound);

**Witnesses** For each argument (operand), a family of languages that ensures that the upper bound is reached (or provides a lower bound);

**References to the literature** The references store a `BIBTEX` entry, for articles on which the result can be found.

An illustrative example, of worst-case state complexity, follows.

**Operation** Catenation

**Argument types** Both arguments must be DFAs over regular languages

**Result type** The result is also a DFA

**Complexity function**  $m2^n - f_12^{n-1}$

**Algorithm** An adaptation of the usual algorithm for catenation of NFAs followed by a specialized subset construction.

**Alphabet size** Must be greater than one

**Restrictions on argument parameters**  $m \geq 1, n > 1, f_1 \geq 1$ , where:

- $f_1$  is the number of final states of the first argument
- $m$  is the number of states of the first argument
- $n$  is the number of states of the second argument

**Witnesses** The following language families ensure the complexity function is reachable:

- For the first argument,  $A = ([0, m - 1], \{a, b, c\}, \delta, 0, \{m - 1\})$ , and for all  $i \in [0, m - 1]$ ,

$$\delta(i, X) = \begin{cases} (i + 1) \bmod m, & \text{if } X=a \\ 0, & \text{if } X=b \\ i, & \text{if } X=c \end{cases}$$

- For the second argument,  $B = ([0, n - 1], \{a, b, c\}, \delta, 0, \{n - 1\})$ , and for all

$$i \in [0, n - 1],$$

$$\delta(i, X) = \begin{cases} i, & \text{if } X=a \\ (i + 1) \bmod n, & \text{if } X=b \\ 1, & \text{if } X=c \end{cases}$$

References to the literature [YZS94]

### 3.6 Language Families

To show that a certain operational complexity bound can be reached, examples of language families  $L_n$ , where  $n$  is related to a complexity measure, must be given. These language families can be described extensionally by a parameterized condition, such as  $L_m = \{x \in \{a, b\}^* \mid \#_a(x) = 0 \bmod m\}$ . These languages can also be defined by models of computation that recognize them, for instance  $L_m$  can be defined by the family of minimal DFAS represented in Fig. 3.1, or by the following regular expression,  $(b^* + (a(b^*a)^{m-1}))^*a(b^*a)^{m-2}b^*$ .

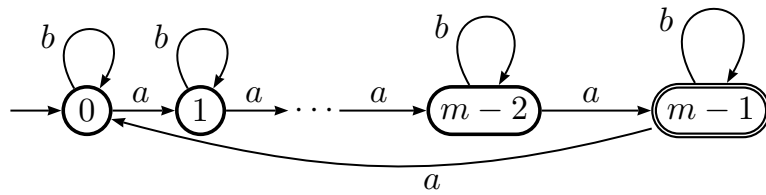


Figure 3.1: A family of minimal DFAS recognizing  $L_m$

It is also possible to represent DFAS using *transformations* [Pin95]. A *transformation* of a set  $Q$  is a mapping of  $Q$  into itself. In general, a transformation has the form:

$$t = \begin{pmatrix} 0 & 1 & \dots & n-2 & n-1 \\ s_0 & s_1 & \dots & s_{n-2} & s_{n-1} \end{pmatrix}$$

Each DFA naturally induces, for each symbol of the alphabet, a transformation on its set of states. Without loss of generality, we assume that the set of states  $Q = \{0, 1, \dots, n-1\}$ , and that the alphabet  $\Sigma = \{0, 1, \dots, k-1\}$ . We can describe a DFA by the transformations induced on its set of states, its initial state and set of final states. For example, the DFA  $\mathcal{A}$ , presented in Fig. 3.2, can be described by its initial state 0, its set of final states  $\{1, 2\}$ , and its transformations  $t_0$  and  $t_1$ :

$$t_0 = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \end{pmatrix}, t_1 = \begin{pmatrix} 0 & 1 & 2 \\ 2 & 2 & 2 \end{pmatrix}$$

Note that the “first” line of each transformation is always the sequence  $0, \dots, n-1$ , so it can be omitted, and thus  $t_0$  becomes  $(1 \ 0 \ 1)$  and  $t_2$  becomes  $(2 \ 2 \ 2)$ .

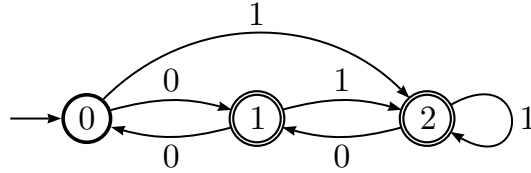


Figure 3.2: DFA  $\mathcal{A}$

This concept can be used to specify families of DFAs in a clean and compact way. The family  $L_m$ , for example, is represented in the following way:

$$m; m; 2; 0; [m-1]; [1..m-1, 0]; [0..m-1]$$

The first field,  $m$ , is the families parameter. The second field,  $m$ , is the number of states each instance must have. The third and fourth fields are the alphabet size

and initial state, respectively, followed by a list of final states, and finally, a list of transformations for each symbol of the alphabet. In this case, since  $|\Sigma| = 2$ , we have two lists, that describe the transition function. This first transformation states the following:

$$\delta(i, 0) = i + 1 \mod m, \forall i \in Q$$

And the second transformation describes the identity function, i.e.,

$$\delta(i, 1) = i, \forall i \in Q$$

In general, a family of DFAS, can be described using the following syntax:

$$p; n; k; i; F; T$$

where,

- $p$  is the families parameter, that can be used in other components,
- $n$  is the number of states,
- $k$  is the alphabet size,
- $i$  is the initial state,
- $F$  is the set of final states,
- $T$  is the set of transformations.

This notation can also be used to specify families of incomplete DFAS, i.e.,  $\exists_{p,s}$  such that  $\delta(p, s)$  is not defined, considering the convention that, if a transformation maps a state  $p \in Q$  to a state  $q \notin Q$ , by a symbol  $s \in \Sigma$ , then the transition function is not defined for state  $p$ , by symbol  $s$ .

This representation, is compact and easy to manipulate and instantiate, and an attempt at standardizing how families of DFAS are represented.

### 3.6.1 The Universal Witness

The notion of *Universal Witness* was introduced by Brzozowski [Brz12]. This notion aims to characterize families of languages that may be more difficult to handle. A family of languages  $L_n$  is said to be universal if it satisfies the following properties:

- A0: The state complexity of  $L_n$  should be  $n$ .** In other words, each instance should be minimal.
- A1: The state complexity of each quotient of  $L_n$  should be  $n$ .** Every strongly connected DFA defines a language that satisfies this condition.
- A2: The number of atoms of  $L_n$  should be  $2^n$ .** An *atom* [BT12] of a regular language  $L$  with quotients  $K_0, \dots, K_{n-1}$ , is a non-empty intersection of the form  $\widetilde{K_0} \cap \dots \cap \widetilde{K_{n-1}}$ , where  $\widetilde{K_i}$  is either  $K_i$  or  $\overline{K_i}$ . The number of atoms of a language is bounded by  $2^n$ , and it was proved that this bound is tight.
- A3: The state complexity of each atom of  $L_n$  should be maximal.** Since every quotient of  $L_n$  is a union of atoms, the atoms of  $L_n$  are its basic building blocks and so they should be as complex as possible.
- A4: The syntactic semigroup of  $L_n$  should have cardinality  $n^n$ .** The syntactic semigroup [Pin97] of  $L$  is the quotient semigroup  $\Sigma^+ / \approx_L$ , where  $\approx_L$  is the Myhill congruence [Myh57], defined as follows:

$$\text{for } x, y \in \Sigma^*, x \approx_L y \text{ if and only if } uxv \in L \Leftrightarrow uyv \in L \text{ for all } u, v \in \Sigma^*$$

Since there are  $n^n$  possible transformations of a set of  $n$  elements,  $n^n$  is an upper bound on the size of the syntactic semigroup of  $L_n$ . That this is a tight bound, was noted by Maslov [Mas70]. This property implies properties **A1**, **A2**, **A3** and **B1**.

**B1: The state complexity of the reverse of  $L_n$  should be  $2^n$ .** It was shown by Mirkin [Mir66] that this upper bound can be reached.

**B2: The state complexity of the star of  $L_n$  should be  $2^{n-1} + 2^{n-2}$ .** It was noted by Maslov [Mas70] that this upper bound is tight, and later proved by Yu, Zhuang and Salomaa [YZS94].

**C1: The state complexity of  $K_m \circ L_n$  should be  $mn$ ,** where  $\circ$  is one of the boolean operations *union*, *symmetric difference*, *intersection*, or *difference* and  $K_m$  is also universal, obtained by permutating the symbols of the alphabet used for  $L_n$ . Yu, Zhuang and Salomaa [YZS94] proved that the upper bound is tight for union and intersection, and Brzozowski [Brz10] for symmetric difference and difference.

**C2: The state complexity of the product  $K_m L_n$  should be  $(m-1)2^n + 2^{n-1}$ .** Yu, Zhuang and Salomaa [YZS94] proved that this upper bound is tight.

Brzozowski gave an example of such a family  $U_n$  which can be described by the following family of DFA  $\mathcal{U}_n$ :

For  $n \geq 3$ , let  $\mathcal{U}_n = \mathcal{U}_n(a, b, c) = (Q, \Sigma, \delta, q_0, F)$ , where  $Q = \{0, \dots, n-1\}$  is the set of states,  $\Sigma = \{a, b, c\}$  is the alphabet,  $q_0 = 0$  is the initial state and  $F = \{n-1\}$  is the set of final states. The transition function  $\delta$  is defined as  $\delta(q, a) = q+1 \pmod n$ ,  $\delta(0, b) = 1$ ,  $\delta(1, b) = 0$ ,  $\delta(q, b) = q$  if  $q \notin \{0, 1\}$ ,  $\delta(n-1, c) = 0$ , and  $\delta(q, c) = q$  if  $q \neq n-1$ .

Using the representation discussed in Section 3.6, the family  $\mathcal{U}_n$  can be seen as:

$$n; n; 3; 0; [n-1]; [1..n-1, 0]; [1, 0, 2..n-1]; [0..n-2, 0]$$

In other words, the transition function, for the first symbol, performs a cycle over all states, for the second symbol it performs a transposition of states 0 and 1, and for the

third symbol it performs a singular transformation sending the state  $n - 1$  to 0. The family of DFAs  $\mathcal{U}_n$  is presented in Fig. 3.3.

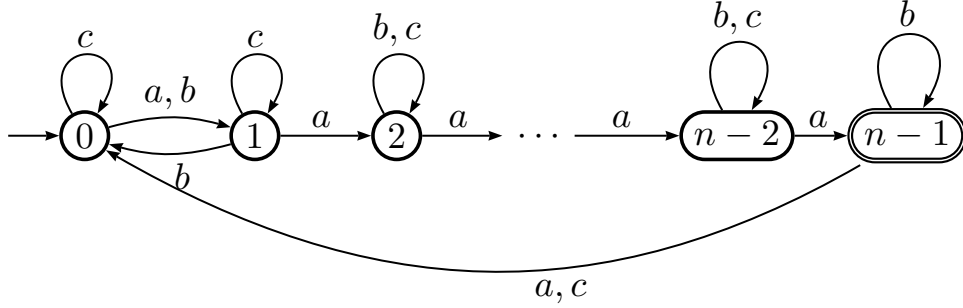


Figure 3.3:  $\mathcal{U}_n(a, b, c)$

The notion of universal witness was not enough to meet the upper bound for some combined operations, and so it was extended to include *dialects* and quaternary alphabets. A dialect of  $U_n$  is the language of any DFA with  $\Sigma = \{a, b, c\}$ , where  $a$  performs a cycle of length  $n$ ,  $b$  transposes any two states and  $c$  performs any singular transformation. To extend the universal witness to alphabets with four symbols, the transition function is defined to perform the identity transformation for the fourth symbol.

With this characterization, it was implemented in DESCO the functionality to operate several instances of the universal witness, its permutationally equivalent languages and its dialects. This feature can be useful to the community when searching for witnesses to meet their upper bounds.

### 3.7 Conclusion

The most important concepts represented in the knowledge base were described in this chapter. Some definitions not covered in Chapter 2 were given. We presented a



new representation for languages families which are compact and easy to manipulate and instantiate. The universal witness and its extensions were also covered.

# Chapter 4

## Desco Components

The DesCo system is implemented in Python [Pyt12], and has four main components: a Web framework, an interface to relational databases, a mathematical typesetter and a formal language symbolic manipulator. In this chapter, we describe each of the above mentioned components and briefly explain how they can be used.

### 4.1 Web Framework

Pylons [Gar11] is an open source Web application framework written in Python, that makes extensive use of the Web Server Gateway Interface [Eby11] standard to promote re-usability and to separate functionality into distinct modules. In the past, developers typically wrote Web applications as a series of simple CGI scripts, each of which would be responsible for accessing the database and generating HTML to produce the pages it outputs. Although each individual script was quick to write and easy to understand, there are a few disadvantages with this approach. Every script in the site needs the same code to load configurations and to handle errors. CGI scripts can be quite slow,

since the whole Python interpreter, as well as, the modules the scripts uses have to be loaded into memory on each request. Also, it can be difficult to understand how the application is structured because each script is almost autonomous. To address these problems, Pylons (as well as other popular frameworks such as Django [JW11], TurboGears [DR11] and Ruby on Rails [HT11]) use a Model View Controller (MVC) architecture.

**The MVC Architecture** is a result of the recognition that most web applications:

- Store and retrieve data (the model)
- Represent data, usually as HTML pages (the view)
- Execute code to manipulate the data and control how it is interacted with (the controllers)

In Pylons each of these components is kept separate. Requests are directed to a controller, which is a Python class that handles the application logic. The controller then interacts with the model classes to fetch data from the database. Once all the information has been gathered, the controller passes this information to a view template where an HTML representation of the data is generated and sent to the user's browser.

## 4.2 Database Interaction

SQLAlchemy [Cop08] is a Python library created to provide a high level interface to relational databases such as PostgreSQL [Gro11], MySQL [AB11] and SQLite [Hip11]. It allows the mapping of Python objects to database tables. For instance, consider the following (MySQL output) table for models of computation:

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
abbreviation	varchar(10)	NO	PRI		
name	varchar(255)	YES		NULL	
description	text	YES		NULL	
FAdoClass	varchar(80)	YES		NULL	

This table can be mapped to the following Python object:

```
class Model(object):
    def __init__(self, abbreviation, name,
                  description, FAdoClass):
        self.abbreviation = abbreviation
        self.name = name
        self.description = description
        self.FAdoClass = FAdoClass
```

This approach is very convenient, as every object instance corresponds to a table record, and this object can have any number of methods implemented, this makes it very easy to manipulate table records. Using SQLAlchemy also has the advantage of not binding the applications source code to a particular database. If we decide that MySQL does not suit our needs, changing to PostgreSQL, for example, only requires that we tell SQLAlchemy to use a different DB-API driver.

## 4.3 Mathematical Typesetting

Since DESCO must deal with a lot of mathematical notation, due to the nature of the system, the ability to, on the one hand, typeset and display math efficiently, and on the other, allow the use of a language that is familiar to the end-user, is crucial. L<sup>A</sup>T<sub>E</sub>X is, arguably, the most popular language, amongst computer scientists, to write mathematical notation, and, as opposed to MathML, it is intended to be written and edited directly by humans. Note also that most of the mathematical formulae in DESCO are functions over integer variables (complexity measures) that can be easily coded in any programming language if any other manipulation is needed.

Listing 4.1: MathJax example

---



---

```
<script src='http://cdn.mathjax.org/
mathjax/latest/MathJax.js?config=default'> </script>
$$
\delta(i,X) = \left \{
\begin{array}{ll}
(i+1)\bmod m, & \text{if } X=a \\
0, & \text{if } X=b \\
i, & \text{if } X=c
\end{array}
\right .
$$.
$$.

```

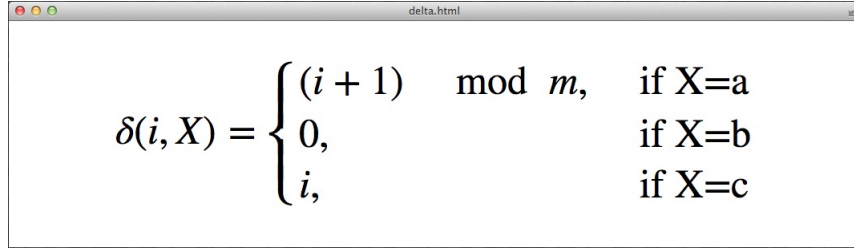
---



---

All this motivates the use of MathJax [Sci11] for the purpose of displaying math. MathJax is a collection of JavaScript programs and support files for displaying mathematics in HTML pages. It works on all modern browsers and does not require any special downloads by the end-user (unlike MathML). It uses HTML/CSS and unicode

fonts for high quality typesetting that is scalable and prints at full resolution. MathJax can display mathematical notation written in  $\text{\LaTeX}$  or MathML markup. However, since it is only meant for math display, only the subset of  $\text{\LaTeX}$  used to describe mathematical notation is supported. One of its most notable features is its ease of use. Say you want to display a transition function, with a branch defined for each symbol of the alphabet on a web page. The page source only needs to load MathJax and have the functions  $\text{\LaTeX}$  source between  $\$$ . The code in Listing 4.1, generates a page, which is exhibited in Fig. 4.1.



$$\delta(i, X) = \begin{cases} (i + 1) \bmod m, & \text{if } X=a \\ 0, & \text{if } X=b \\ i, & \text{if } X=c \end{cases}$$

Figure 4.1: Web browser displaying a MathJax generated formula

## 4.4 Formal Language Symbolic Manipulator

The FADO system [AAA<sup>+</sup>09] aims to provide an open source extensible software library for the symbolic manipulation of automata and other models of computation. To allow high-level programming with complex data structures, easy prototyping of algorithms, and portability, are its main features. FADO is implemented in Python [Pyt12] and currently includes most standard operations for the manipulation of regular languages. Regular languages can be represented by regular expressions (REEX) or finite automata, among other formalisms. Finite automata may be deterministic (DFA), non-deterministic (NFA) or generalized (GFA). In FADO, these representations are implemented as Python classes. Elementary regular language operations, such as,

union, intersection, concatenation, complementation, and reverse are implemented for each class. Many combined operations for DFA have specialized algorithms. Several conversions between representations are implemented: NFA to DFA via subset construction, NFA to REEX using a recursive method, GFA to REEX using the state elimination algorithm, with possible choice of state orderings and several heuristics, REEX to NFA using the Thompson method, Glushkov method, follow, and partial derivatives. For DFA, several minimization algorithms are available: Moore, Hopcroft, Brzozowski, and some incremental algorithms. Some support is provided for computing syntactic semigroups. There are several algorithms for language equivalence. Finite languages can also be represented, by tries and AFA (acyclic finite automata).

Symbolic manipulation systems, and the FADO system in particular, are essential for the studies of descriptonal complexity, as they provide a tool for testing the complexity bounds and to find candidate witnesses. FADO implements many of the specialized algorithms used for the upper bounds of the operational complexities of regular languages and also many of the language families used as witnesses.

The FADO system currently aids DESCo in the generation and manipulation of language family instances. Either by using the representation using transformations, as previously defined in Section 3.6, or by using Python code, we can generate language family instances. Once a FADO class, representing a model of computation, has been instantiated, we can then make use of all the algorithms it has to offer to do symbolic manipulation, such as checking whether or not the first few instances of a family of DFAs are minimal, checking the equivalence of several instances of two different language families, or even verify if the given witnesses reach the upper bound for specific parameters of the language family. It is also possible to guess whether a language family is already in the knowledge base. By instantiating a family, for

example by varying its parameter between 3 and 10, then converting each instance to its *canonical string representation* [AMR07], and concatenating all the strings, we obtain a representation which would allow a good guess to be made when comparing language families. To compare two language families, one only has to compare the obtained strings, and if they are the same then there is a high chance that the language families are the same.

## 4.5 Conclusion

This chapter described the functionality of the four main components of the DESCO system: *Pylons* the web framework, *SQLAlchemy* the interface to the relational database, *MathJax* the mathematical typesetter and FADO the formal language symbolic manipulator. Some small examples of how they can be used were given, as well as, some uses of the formal language manipulator.



# Chapter 5

## The Web Interface

The web interface allows the manipulation and visualization of all the existing results and respective bounds, as well as, all the concepts described in Chapter 3. Its source code is generated using Mako, a Python templating language. MathJax is used for mathematical typesetting and FADO is used as a symbolic manipulator when such functionality is requested by the user. The structure and functionality of the web interface is described in this chapter. Introducing and updating information is discussed in Section 5.1, the various ways to consult the knowledge base are described in Section 5.2, and how the user can interact with FADO through DESCo is covered in Section 5.3.

### 5.1 Introducing and Updating Information

Inserting and updating data through the web interface is done using a series of forms built with common HTML input elements, such as, text boxes, check boxes, radio buttons, etc., and rich text editors. Fig. 5.1 shows the form for adding a

new operation. Such tasks can only be performed by authenticated users, and the information submitted can be alter, if necessary, at a later time by any authenticated user that has updating privileges. The main page provides quick access to the forms for adding languages classes, operations, models and their variables specifications, complexity kinds and measures, language families and bibtex entries.

### New operation

---

Name	<input type="text" value="Circle"/>	?	
LateX Symbol	<input type="text" value="L_1\circ L_2"/>	?	
	Preview: $L_1 \circ L_2$		
FAdo Function	<input type="text"/>	?	
Combined	<input type="checkbox"/>		
Arity	<input type="text" value="2"/>		
Description ?	<div style="border: 1px solid #ccc; padding: 5px;"> <div style="display: flex; justify-content: space-between; align-items: center; border-bottom: 1px solid #ccc; padding-bottom: 5px;"> <div> <span style="border: 1px solid #ccc; padding: 2px 5px; margin-right: 5px;">B</span> <span style="border: 1px solid #ccc; padding: 2px 5px; margin-right: 5px;">I</span> <span style="border: 1px solid #ccc; padding: 2px 5px; margin-right: 5px;">↔</span> <span style="border: 1px solid #ccc; padding: 2px 5px; margin-right: 5px;">⋮</span> <span style="border: 1px solid #ccc; padding: 2px 5px; margin-right: 5px;">⋮</span> <span style="border: 1px solid #ccc; padding: 2px 5px; margin-right: 5px;">🖼</span> <span style="border: 1px solid #ccc; padding: 2px 5px; margin-right: 5px;">🔍</span> <span>Change block type ▾</span> </div> <div style="flex-grow: 1;"> <p>This is an example of an operation <math>L_1 \circ L_2</math> which could be defined as follows...</p> </div> </div> </div>		

Figure 5.1: Form for adding a new operation

Since some concepts have natural dependencies between them, it is possible that while introducing new information, some parts of the form can not be filled. For example, if we attempt to introduce a new complexity measure, *state complexity*, we must first make sure that its associated model of computation, DFA, exists in our knowledge base. If DFA is not in the knowledge base, the user would be unable to fill out that particular item of the form. This is not a problem however, as the user can simple go back, introduce this missing concept, and then introduce the new complexity measure.

Some of the forms used for introducing and updating information are equipped with a preview functionality in order to avoid mistakes, which during the development of

DESCO became quite obvious they would be very common. For instance, some forms require the user to use  $\text{\LaTeX}$  markup language. In such cases it was not obvious whether the input had to be enclosed in \$ signs. With the preview functionality this requirement now becomes obvious for the user.

## 5.2 Consulting the Knowledge Base

Consulting the knowledge base, in contrast to inserting and updating data, does not require any kind of authentication, thus the information is available to everyone. Querying the knowledge base on language classes, language families, operations, models of computation, complexity kinds or complexity measures results in a table listing the requested information. This listing is formatted using HTML/CSS and MathJax for a more aesthetically pleasing visualization. An example of such a listing is illustrated in Fig. 5.2. The main page provides such listings.

**Reversal -  $L^R$**

Arity: 1

For a word  $x$  over an alphabet  $\Sigma$ , the *reversal* of  $x$  is denoted by  $x^R$  and it is recursively defined by:

$\epsilon^R = \epsilon$ , and  $(ay)^R = y^R a$  where  $a \in \Sigma$  and  $y \in \Sigma^*$ .

By definition, if  $x = a_1 \dots a_n$ , where  $n \geq 1$  and  $a_1, \dots, a_n$  are letters in  $\Sigma$ , the  $x^R = a_n \dots a_1$ .

For a language  $L$  over an alphabet  $\Sigma$ , the *reversal* of  $L$  is denoted by  $L^R$  and is defined by  $L^R = \{x^R \mid x \in L\}$ .

**Star -  $L^*$**

Arity: 1

The *Kleene star* (also known as *Kleene operator* or *Kleene closure*) of a language  $L$  can be defined as:

- $L^* = \{x_1 x_2 \dots x_n \mid n \geq 0 \text{ and } x_i \in L, 1 \leq i \leq n\}$

Given  $A$  and  $B$  languages, the following properties hold for the Kleene star operation.

Figure 5.2: Listing operations

Language families can be further queried for more information, such as, the code that generates its instances. Additionally, a diagram that represents a given language family can be generated on the fly. This is exemplified in Fig. 5.3.

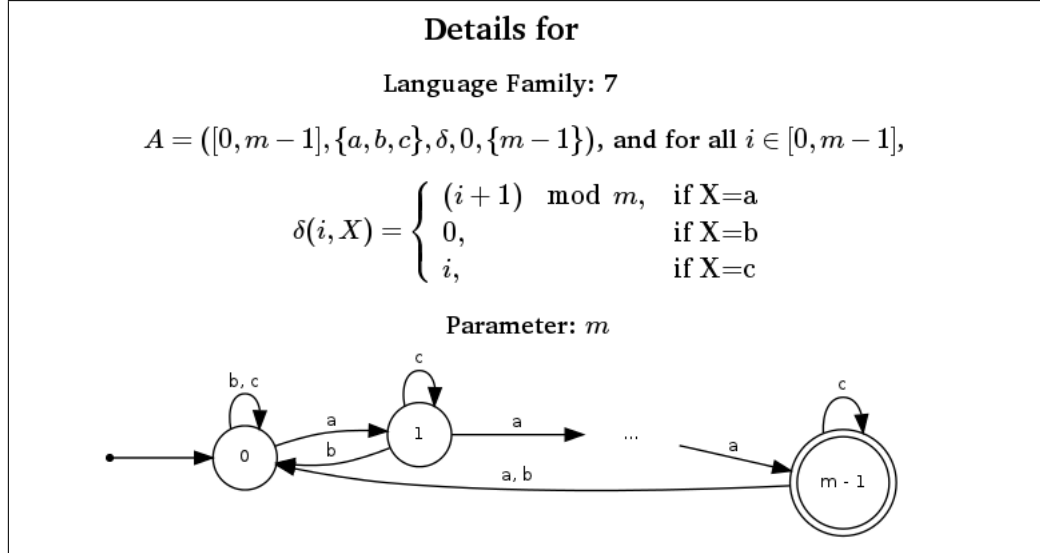


Figure 5.3: Language family details

The complexity bounds can be consulted with more fine-grain detail, giving control over how the information is displayed by customizing query parameters, using the *DataBase Viewer*. Firstly, one must choose the complexity kind. Then, one can choose to query all the results for a single operation. This generates a table with the complexity functions for all language classes and all complexity measures, for that specific operation, as seen in Fig. A.1.

On the other hand, one can also choose a list of operations and a specific complexity measure, generating a table for each language class with all the results for the chosen operations, as presented in Fig. 5.4. It is also possible to list results, for all operations, for a given language class and one or more complexity measures.

All the available details, about a specific result, such as its witnesses, references and

algorithms, can also be consulted, as portrayed in Fig. A.2, by selecting the desired result from the table generated by the *DataBase Viewer*.









Worst-Case State Complexity		
Click on a function to get more detailed information		
Finite Language		
$L_1 L_2$	 	$O(mn^{f_1-1} + n^{f_1})$ , if $f_1 > 0$ for $ \Sigma  > 1$ $(m - n + 3)2^{n-2} - 1$ , if $m + 1 > n > 2$ for $ \Sigma  = 2$
$L_1 \cap L_2$	 	$mn - 3(m + n) + 12$ for $ \Sigma  = mn - 3(m + n) + 11$
$L^*$	 	$m^2 - 7m + 13$ , if $m > 4, f \geq 3$ for $ \Sigma  = 1$ $2^{m-3} + 2^{m-l-2}$ , if $l \geq 2, m \geq 4$ for $ \Sigma  > 2$
Regular Language		
$L_1 L_2$	 	$m$ , if $m > 0, n = 1$ for $ \Sigma  > 0$ witnesses for: $m > 0, n = 1$ $m2^n - f_1 2^{n-1}$ , if $m \geq 1, n > 1, f_1 \geq 1$ for $ \Sigma  > 1$ witnesses for: $m = 1, n \geq 2; m \geq 2, n \geq 2$

Figure 5.4: Listing results for several operations and language classes

## 5.3 Interacting with FAdo

FADO plays an important role in DESCo. It is the symbolic manipulator used to operate language family instances. If the knowledge base contains the FADO function, which given representations of languages as arguments returns the resulting languages after performing an operation, as well as a representation of the language families to be used as arguments, then DESCo can make use of FADO to generate its instances, perform the given operation and return the result in several formats. The available formats are *.svg*, *.fa* and plain information about the result. The *.fa* format allows the user to download the result and use it later on with FADO, on their own machine. The *.svg* format checks if the resulting automaton has less than 150 states, and if so, displays the diagram associated to the automaton on a browser window, otherwise, the user can download the *.dot* source code (a popular plain text graph description

language) and convert it on their own machine later on, in order to not overload the server.

Another way of interacting with FADO through DESCO is by performing operations using the *Universal Witness* described in Section 3.6.1. The user can choose from any of the operations that have a FADO function associated in the knowledge base and parameterize the arguments in order to define which dialect and permutation of the alphabet to use. For each argument, its parameter is instantiated for all integer values in a given interval, the operation is computed on the server and the information about the result is then displayed to the user. This feature can be useful to the community when searching for witnesses to meet their upper bounds.

## 5.4 Conclusion

This chapter describes how information can be introduced, updated and consulted through the web interface. Some features that the forms for introducing and updating information are equipped with were described, as well as, how the user can consult the knowledge base in various ways. How the user can interact with FADO through DESCO was also outlined.

# Chapter 6

## Conclusion

This thesis describes the DESCO system, a Web-based knowledge system for descriptonal complexity results. It gives an introduction to formal languages and descriptonal complexity, characterizes the key concepts involved and describes how they can be represented in a knowledge base. Some of these concepts, namely language families, require careful consideration, in order to obtain a representation that can be manipulated in such a way as to help the research community with their work in descriptonal complexity.

The tools employed in the creation of the knowledge base, web interface and symbolic manipulation were delineated, as well as, the functionality implemented in the web interface.

### 6.1 Current State of DesCo

DESCO is currently online ([desco.up.pt](http://desco.up.pt)) and an authenticated user can also submit new information. Most of the data accessible is related with the operational complexity

of regular or subregular languages, with over three hundred results and more than sixty language families already available. Even for this subset of descriptonal complexity results, it is now easier to have a general overview, analyze the complexity behavior of different operations for a language class, or compare the relative complexity of the same operation on different language classes.

One motivation for this project was to help ease the work of anyone refereeing for a conference or journal in the field of descriptonal complexity. Even though there is a considerable amount of information still missing from the knowledge base, it is now possible, with the help of DESC<sub>O</sub>, to look up specific results and their references to the literature. This can be of great help to the community. Also, it is now easier for members of the research community to decide where to allocate their efforts in the field of descriptonal complexity.

## 6.2 Future Work

Possible future work could include allowing a test of universality for language families, i.e., test whether a language family satisfies the properties enumerated in Section 3.6.1. It may not be possible to test all of the properties, since some of them might require the notion of dialects to satisfy, which are not defined for all language families.

To prove that a complexity bound is tight there are also other techniques besides the use of language families, for instance the *fooling-set lower bound technique* [GS96]. That also has to be accommodated in DESC<sub>O</sub>.

Even though the DESC<sub>O</sub> system was planned to be as flexible as possible, in order to accommodate any result in descriptonal complexity, as the system evolves and new results are included, new concepts may arise that are currently not supported.



Such problems are likely to come up and have to be dealt with in the future. Finally, the continued usage of the system by the community, will likely lead to more improvements.

# Appendix A

## Web Interface Screenshots

Worst-Case for Union		
Language	sc	nsc
Sortable	Resizable	Resizable
Prefix-free	$mn - 2$ , if $m, n \geq 3$ $\max\{m, n\}$	$m + n$
Prefix-closed	$\max\{m, n\}$ $mn$	
Finite	$\max\{m, n\}$ $mn - (m + n)$	$m + n - 2$ , if $m \geq 2, n \geq 2$ $\max\{m, n\}$
Regular	$mn$ , if $m > 0, n > 0$	$m + n + 1$

Figure A.1: Listing results for union operation

$sc(L^*) \leq 2^{m-1} + 2^{m-l-1}$ For alphabet of size $> 1$ and $m > 1, l > 0$ Where: <ul style="list-style-type: none"> <li>• <math>l</math> is <math> F - \{q_0\} </math> the number of states that are final but are not the initial state of <math>L_1</math></li> <li>• <math>m</math> is <math> Q </math> the number of states of the minimal DFA of <math>L_1</math></li> </ul>	
Description	Input: $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ , $L_1 = L(A_1)$ and $ F_1 - \{q_1\}  = l_1 \geq 1$ Output: $C = (Q, \Sigma, \delta, q, F)$

Figure A.2: Details for a particular result

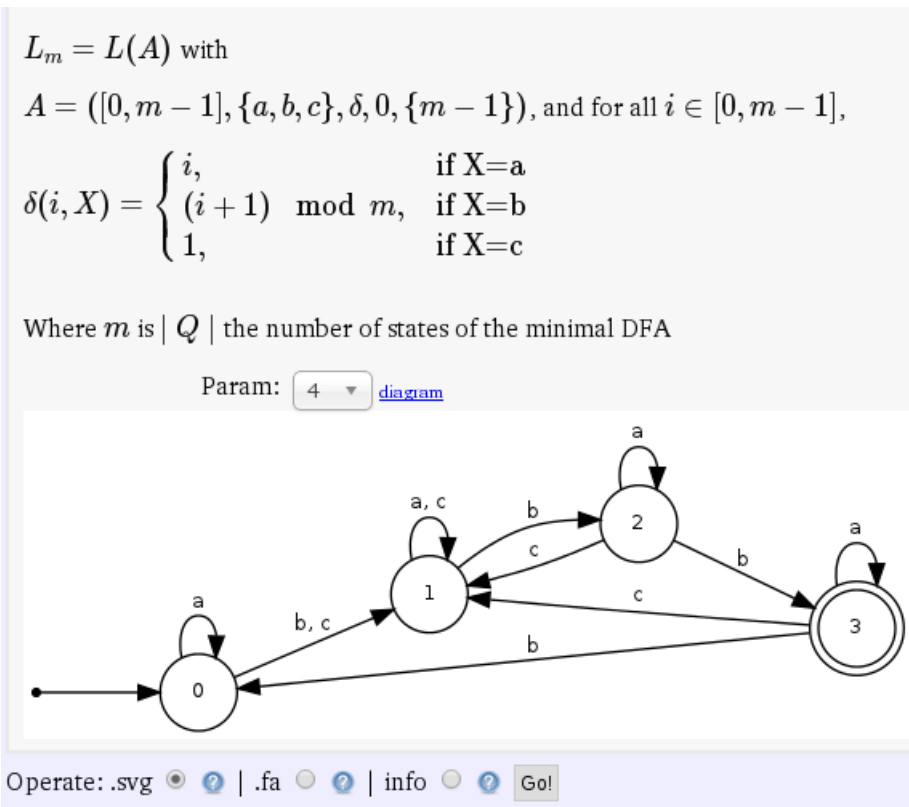


Figure A.3: FAdo generating instances

$n - \text{sigma} - \text{Finals} - (i\_2, j\_2) - (i\_3, j\_3)$

$n;n;3;0;[n-1];[1..n-1,0];[1,0,2..n-1];[0..n-2,0]$

[Export](#) results as csv file.

Inputs	States	Finals	Transitions
3	6	3	14
4	12	5	30
5	24	9	62
6	48	17	126
7	96	33	254

Figure A.4: FAdo operating with universal witnesses

# References

- [AAA<sup>+</sup>09] A. Almeida, M. Almeida, J. Alves, N. Moreira, and R. Reis. FAdo and GUItar: tools for automata manipulation and visualization. In S. Maneth, editor, *14th CIAA Proceedings*, volume 5642 of *LNCS*, pages 65–74, Sidney, July 2009. Springer.
- [Aar11] Scott Aaronson. The Complexity Zoo. [qwiki.stanford.edu/index.php/Complexity/Zoo](http://qwiki.stanford.edu/index.php/Complexity/Zoo), 2011.
- [AB11] MySQL AB. MySQL. [www.mysql.com](http://www.mysql.com), 2011.
- [Alg11a] Algorithms Project. Dynamic Dictionary of Mathematical Functions (DDMF). [ddmf.msr-inria.inria.fr/1.6/ddmf](http://ddmf.msr-inria.inria.fr/1.6/ddmf), 2011.
- [Alg11b] Algorithms Project. Encyclopedia of Combinatorial Structures (ECS). [algo.inria.fr/encyclopedia/intro.html](http://algo.inria.fr/encyclopedia/intro.html), 2011.
- [AMR07] M. Almeida, N. Moreira, and R. Reis. Enumeration and generation with a string automata representation. *Theor. Comput. Sci.*, 387(2):93–102, November 2007.
- [Brz10] Janusz A. Brzozowski. Quotient complexity of regular languages. *Journal of Automata, Languages and Combinatorics*, 15(1/2):71–89, 2010.

- [Brz12] Janusz A. Brzozowski. In search of most complex regular languages. In Nelma Moreira and Rogério Reis, editors, *17th CIAA Proceedings*, volume 7381 of *LNCS*, pages 5–24, Porto, Portugal, 2012. Springer.
- [BT12] Janusz A. Brzozowski and Hellis Tamm. Quotient complexities of atoms of regular languages. In Hsu-Chun Yen and OscarH. Ibarra, editors, *Developments in Language Theory*, volume 7410 of *Lecture Notes in Computer Science*, pages 50–61. Springer Berlin Heidelberg, 2012.
- [Cop08] Rick Copeland. *Essential SQLAlchemy*. O’Reilly Media, 2008.
- [DR11] Kevin Dangoor and Mark Ramm. TurboGears. [turbogears.org](http://turbogears.org), 2011.
- [Eby11] P. J. Eby. Python web server gateway interface v1.0.1. [www.python.org/dev/peps/pep-3333](http://www.python.org/dev/peps/pep-3333), 2011.
- [Gar11] James Gardner. Pylons book, 2011.
- [GKK<sup>+</sup>02] Jonathan Goldstine, Martin Kappes, Chandra M. R. Kintala, Hing Leung, Andreas Malcher, and Detlef Wotschke. Descriptive complexity of machines with limited resources. *J. UCS*, 8(2):193–234, 2002.
- [Gro11] PostgreSQL Global Development Group. PostgreSQL. [www.postgreSQL.org](http://www.postgreSQL.org), 2011.
- [GS96] Ian Glaister and Jeffrey Shallit. A lower bound technique for the size of nondeterministic finite automata. *Inf. Process. Lett.*, 59(2):75–77, 1996.
- [Hip11] D. Richard Hipp. SQLite. [www.sqlite.org](http://www.sqlite.org), 2011.
- [HK11] Markus Holzer and Martin Kutrib. Descriptive and computational complexity of finite automata - a survey. *Inf. Comput.*, 209(3):456–470, 2011.

- [HMU06] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 2006.
- [Hro02] Juraj Hromkovic. Descriptive complexity of finite automata: Concepts and open problems. *J. Aut. Lang. Comb.*, 7(4):519–531, 2002.
- [HT11] David Heinemeier Hansson and Rails Core Team. Ruby on Rails. [rubyonrails.org](http://rubyonrails.org), 2011.
- [JW11] Lawrence Journal-World. Django. [www.djangoproject.com](http://www.djangoproject.com), 2011.
- [Kap12] Christos A. Kapoutsis. minicomplexity. <http://minicomplexity.org>, 2012.
- [Mas70] A. N. Maslov. Estimates of the number of states of finite automata. *Doklady Akademii Nauk SSSR*, 194:1266–1268, 1970. English translation in *Soviet Mathematics Doklady*, 11, 1373–1375 (1970).
- [Mir66] Boris G. Mirkin. On dual automata. *Kibernetika*, 2:7–10, 1966. English translation in *Cybernetics* 2, 6–9 (1966).
- [Myh57] J. Myhill. Finite automata and the representation of events. *Wright Air Development Center Technical Report*, pages 57–62, 1957.
- [Ner58] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.
- [Pin95] Jean-Eric Pin. Finite semigroups and recognizable languages: an introduction. In *NATO Advanced Study Institute*, 1995.
- [Pin97] Jean-Eric Pin. Syntactic semigroups. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, pages 679–746. Springer Berlin Heidelberg, 1997.

- [Pyt12] Python Software Foundation. Python language website. <http://python.org>, 2012.
- [Sci11] Design Science. MathJax. [www.mathjax.org](http://www.mathjax.org), 2011.
- [SP95] N. J. A. Sloane and S. Plouffe. *The Encyclopedia of Integer Sequences*. Academic Press, 1995.
- [The11] The OEIS Foundation. The On-line Encyclopedia of Integer Sequences (OEIS). [oeis.org](http://oeis.org), 2011.
- [Yu05] Sheng Yu. State complexity: Recent results and open problems. *Fundam. Inform.*, 64(1-4):471–480, 2005.
- [YZS94] Sheng Yu, Qingyu Zhuang, and Kai Salomaa. The state complexities of some basic operations on regular languages. *Theor. Comput. Sci.*, 125(2):315–328, 1994.
- [Zol11] Evgeny Zolin. Navigator on description logic complexity. [www.cs.man.ac.uk/~ezolin/dl/](http://www.cs.man.ac.uk/~ezolin/dl/), 2011.