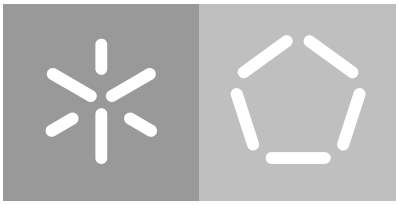


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Paulo Ribeiro

Regular Expressions for Timed Automata

July 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Paulo Ribeiro

Regular Expressions for Timed Automata

Master dissertation
Master Degree in Physics Engineering

Dissertation supervised by
Luis Barbosa
Renato Neves

July 2021

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.



https://creativecommons.org/licenses/by/4.0/

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ACKNOWLEDGEMENTS

I want to thanks my parents, Antonio and Rosa, and my brother, Tiago, for the unconditional support given during this long process. Moreover, all my family for the motivation they provided me.
A special thanks to my co-supervisor, Renato Neves, for having faith in my capacities and never giving up on me throughout this MSc dissertation. Also, to my supervisor Luis Barbosa for all the help.
A big thank you to my friends, Ricardo and Abilio, who supported me and were always there for me.
To my classmates and friends in my course. Thank you all for the great company during this long journey we spent together; I could not ask for better. Good luck in the future.
This work is a result of the project "SmartEGOV: Harnessing EGOV for Smart Governance (Foundations, Methods, Tools) - NORTE-07-0124-FEDER-000037", supported by the Norte Portugal Regional Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, through the European Regional Development Fund (ERDF).
This work is also financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-030947.

ABSTRACT

Finite automata are valuable models for various types of software, such as system verification software and text search software, etc. A widely acclaimed result in computer science is the proof that finite automata can be represented through a notation structure called a regular expression. This means that regular expressions succeed in describing the same patterns that can be represented by a finite automaton. We know that it is difficult to prove that two finite automata are equivalent. Still, by converting them to a regular expression, you can determine relatively easily whether they represent the same language.
Timed automata are an extension of finite automata with a finite set of clocks. Timed automata are widely used in *model checking* and also in real-time systems such as communication and security protocols. In recent years there is an increasing demand for this type of software. As in classical automata, timed automata can also be represented by regular expressions. However, so far, there is no conversion software developed. There are two main methods for converting: the state elimination method; and the Brzozowski's method. These methods refer to the classic automaton model.
The goal of this work is the study and development of an algorithm that converts a timed automaton into a timed regular expression. For this, 1) we developed a conversion algorithm based on Brzozowski's method for timed automata adding to the classical case several changes, in particular, the incorporation of clocks and transition restrictions, among other, 2) we developed a software tool that converts a timed regular expression into a timed automaton, and depicting both of them.

Keywords: Algorithm, Timed Regular Expression, Brzozowski's method, Software tool

RESUMO

Autómatos finitos são modelos úteis para vários tipos de software, tais como, software de verificação de sistemas e software de pesquisa de texto, etc. Um resultado amplamente aclamado na ciência da computação é a prova de que autómatos finitos podem ser representados através de uma estrutura de notação chamada expressão regular. Estas expressões regulares conseguem descrever os mesmos padrões que podem ser representados por um autómato finito. Sabemos que é difícil provar que dois autómatos finitos são equivalentes, mas ao converte-los para uma expressão regular consegue-se determinar com relativa facilidade se eles representam a mesma linguagem.

Os autómatos temporais são uma extensão de autómatos finitos com um conjunto finito de relógios. Os autómatos temporais são muito usados em *model cheking* e também em sistemas de tempo-real, como protocolos de comunicação e segurança. Nos últimos anos tem havido procura crescente para esse tipo de software. Tal como nos autómatos clássicos, os autómatos temporais podem ser representados por expressões regulares. Mas até agora não existe software de conversão desenvolvido. Existem dois métodos principais para efetuar a conversão: método de eliminação de estados; e o método de Brzozowski. Estes métodos funcionam para o modelo clássico de autómato.

O objetivo deste trabalho é o estudo e desenvolvimento de um algoritmo que converta um autómato temporal numa expressão regular. Para isso, 1) desenvolvemos um algoritmo de conversão baseado no método de Brzozowski, incorporando diversos elementos adicionados, nomeadamente, relógios e restrições nas transições, entre outras 2) desenvolvemos uma ferramenta que converte uma expressão regular temporal num autómato temporal, e no final estes dois elementos são representados numa única imagem.

Palavras-chave: Algoritmo, Expressões Regulares Temporais, método de Brzozowski, Ferramenta de software

CONTENTS

1 INTRODUCTION 1
- 1.1 Motivation 1
- 1.2 Context and Main Contribution 1
- 1.3 Document Structure 2

2 A PRIMER ON FINITE AUTOMATA 3
- 2.1 Basic Notions 3
  - 2.1.1 Alphabets and Strings 4
- 2.2 Deterministic Finite Automata 5
- 2.3 Non-Deterministic Finite Automata 6
- 2.4 ω-automata 8
- 2.5 Regular Expressions 10
  - 2.5.1 Operations on Regular Expressions 10
  - 2.5.2 From DFAs to Regular Expressions 12

3 WHEN REAL-TIME COMES INTO SCENE 19
- 3.1 Summary 19
- 3.2 A theory of Timed Automata 19
  - 3.2.1 Timed Languages 20
  - 3.2.2 Timed transition tables 21
  - 3.2.3 Transition tables with timing constraints 22
  - 3.2.4 Clock constraints 23
  - 3.2.5 Timed regular language 23

4 TIMED REGULAR EXPRESSIONS 25
- 4.1 Introduction 25
- 4.2 From timed automata to timed regular expressions (a sketch) 27
- 4.3 The formal definition of timed regular expression & Examples of the conversion method 31

5 IMPLEMENTATION 41
- 5.1 Overview 41
  - 5.1.1 Parser & Parser Utils 42
  - 5.1.2 Regular Expressions 44
  - 5.1.3 Draw 47
  - 5.1.4 Main 48

5.2 Key ideas in the implementation 48
6 CASE-STUDIES 49
6.1 Mutual Exclusion 49
6.2 Railway-Crossing System 51
7 CONCLUSION 58
7.1 Summary 58
7.2 Future work 59

A INSTRUCTION MANUAL FOR INSTALLATION AND USE 63
B EXAMPLES 65
B.1 Level 1 65
B.1.1 Example 1 65
B.1.2 Example 2 65
B.1.3 Example 3 66
B.2 Level 2 66
B.2.1 Example 1 66
B.2.2 Example 2 67
B.2.3 Example 3 67
B.2.4 Example 4 67
B.3 Level 3 68
B.3.1 Fischer protocol 68
B.3.2 Controller 68
B.3.3 Gate 69
B.3.4 Train 69

LIST OF FIGURES

Figure 1	Simple Finite Automaton with inputs 0/1.	3
Figure 2	Finite Automaton recognizes the word "cut".	4
Figure 3	Non-Deterministic Finite Automaton with 2 states	5
Figure 4	Non-Deterministic Finite Automaton with 3 states	5
Figure 5	Deterministic Finite Automaton	6
Figure 6	Non-Deterministic Finite Automaton	6
Figure 7	The states of an NFA is during the processing the input abb	7
Figure 8	Deterministic Finite Automaton	8
Figure 9	Büchi Automaton recognizing $(a+b)a^{\omega}$	9
Figure 10	An DFA accepting the string ab	13
Figure 11	Eliminating state q_1	13
Figure 12	Generic two-state automaton	13
Figure 13	Generic one state automaton	14
Figure 14	An NFA that accepts strings that begin with a and end with either b or ab	14
Figure 15	The automaton of Fig. 14 with different labels	14
Figure 16	Eliminating state q_1	15
Figure 17	Eliminating state q_3	15
Figure 18	A two-states automaton with states q_0 and q_4	15
Figure 19	NFA which accepts strings that begin with a and end with either b or ab	17
Figure 20	Timed transition table	23
Figure 21	Timed Büchi Automaton	24
Figure 22	Description of all components of a Timed Regular Expression	27
Figure 23	A Timed Regular Expression	27
Figure 24	A timed automaton with three state	28
Figure 25	A timed automaton with three states	33
Figure 26	A timed automaton with three state generated through our algorithm	33
Figure 27	A timed regular expression	33
Figure 28	A timed regular expression generated through our algorithm	33
Figure 29	A timed automaton with a single state	34

Figure 30 A timed automaton with a single state generated through our algorithm 34
Figure 31 A timed regular expression 34
Figure 32 A timed regular expression generated through our algorithm 34
Figure 33 A timed automaton with three states 35
Figure 34 A timed automaton with three state generated through the algorithm 35
Figure 35 A timed regular expression 35
Figure 36 A timed regular expressions generated through our algorithm 35
Figure 37 A timed automaton 36
Figure 38 A timed automaton generated through our algorithm 36
Figure 39 A timed regular expression 37
Figure 40 Timed automaton regular generated through our algorithm 37
Figure 41 A Timed Automaton with four states 37
Figure 42 A timed automaton with four states generated through our algorithm 37
Figure 43 A timed regular expression 38
Figure 44 A timed regular expression generated through our algorithm 38
Figure 45 A timed automaton with four states 38
Figure 46 A timed automaton with four state generated through the algorithm 38
Figure 47 A timed regular expression 39
Figure 48 A timed regular expression generated through the algorithm 39
Figure 49 A timed automaton with four state 39
Figure 50 A timed automaton generated through our algorithm 40
Figure 51 A timed regular expression 40
Figure 52 A timed regular expression generated through our algorithm 40
Figure 53 Error message 43
Figure 54 Timed automaton 45
Figure 55 Diagrammatic description of the regular expressions 47
Figure 56 File manager 48
Figure 57 Mutual Exclusion timed automaton 50
Figure 58 Mutual Exclusion timed automaton generated 50
Figure 59 Mutual Exclusion timed regular expression 51
Figure 60 Mutual Exclusion timed regular expression generated 51
Figure 61 Train timed automaton 52
Figure 62 Train timed automaton generated 52
Figure 63 Train timed regular expression 53
Figure 64 Train timed regular expression generated 53

Figure 65	Gate timed automaton	54
Figure 66	Gate timed automaton generated	54
Figure 67	Gate timed regular expression	55
Figure 68	Gate timed regular expression generated	55
Figure 69	Controller timed automaton	56
Figure 70	Controller timed automaton generated	56
Figure 71	Controller timed regular expression	56
Figure 72	Controller timed regular expression generated	57

LIST OF TABLES

Table 1 Transition table of the automaton in Fig. 8 8

1

INTRODUCTION

1.1 MOTIVATION

Kleene's theorem [Kleene (1956)] is one of the most famous results in computer science. It establishes a correspondence between deterministic finite automata and regular expressions, showing that one can always be converted into the other. More specifically, it proves that a language over an alphabet is regular iff it can be accepted by an automaton. This work was followed by several generalizations and also by the introduction of a *complete axiomatization for regular expressions* [Salomaa (1966); Kozen (1994)], which provides a powerful and a concise way to think about deterministic automata.
A few years after Kleene's work, Alur and others introduced the notion of *timed automata* [Alur and Dill (1994)]. It was one of the first models to deal with *real-time systems*. With these models, we can have better control over physical processes, e.g. speed, movement, energy, and time, essential to deal with an increasing demand for this kind of software development. Nevertheless, up until now, there is no software to convert a *timed automata* to a regular expression, understood as a structural notion for describing the same patterns that can be represented by finite automata [Hopcroft et al. (2007)].
As we know, automata are very useful in various search problems. However, it is hard to determine if two automata are equivalent. Regular expressions bring notion of equivalence, and expanding it into the timed case can be an asset. Therefore, we propose to develop an algorithm that converts a timed automaton into a timed regular expression.

1.2 CONTEXT AND MAIN CONTRIBUTION

A Kleene's theorem for timed automata (a subclass of hybrid automata) was proved in Asarin et al. (1997), but a complete axiomatisation remains to be developed. Then, in the

work of Neves and Barbosa (2016), they showed that hybrid automata and several of their variants could be uniformly organised and classified within a *coalgebraic* framework [Rutten (2000)], which among other things, provided a Kleene's theorem for different types of hybrid automata. Similarly to Asarin et al. (1997), complete axiomations were not developed nor computational tools to support these results. In the current dissertation, we take advantage of these results to implement a tool that converts timed automata into 'timed regular expressions'.

1.3 DOCUMENT STRUCTURE

The current dissertation is divided into seven sections. Section 1 motivates our work and main contributions. Section 2 provides the necessary background, a general view of classical automata and regular expressions. Our overview is will mostly base on Hopcroft et al. (2007). Section 3 provides some general concepts on the theory of timed automata. In Section 4 we present the theoretical basis of our main contribution. We begin by introducing the differences between our approach and the concepts in sections two and three on converting automata to regular expressions. Then we recall a definition of a *timed automata* and *timed regular expression* and illustrate both cases via different examples. In Section 5 we introduce the implementation of the theory developed in Section 4, particularly describing its architecture and main features. In Section 6 we present case studies that illustrate our tool at work. Section 7 conclusions and some possible lines of future work. In the appendix, we provide some examples for the user and installation of the program developed in the dissertation.

2

A PRIMER ON FINITE AUTOMATA

2.1 BASIC NOTIONS

The notion of finite state machines was formulated in the 1930s by A. Turing. It was the first introduction of how a computer could work, which furthermore established a notion of computability. In his work, Turing also showed that undecidable problems exist, i.e. their solutions, cannot be effectively obtained by computational mechanisms [Aho et al. (2006); Eden (2007)].

An automaton is a simpler version of a Turing machine; specifically, it is simply a machine comprised of a set of states that can switch between them. The letter can be seen as a representation of internal memory.

Example 2.1.1. *Consider a simple automaton with two states q_1 and q_2 and two arcs labelled with 0 and 1. The automaton has two possible transitions: $q_1 \to q_2$ if it receives an input 0 and $q_2 \to q_1$, if receives an input 1.*

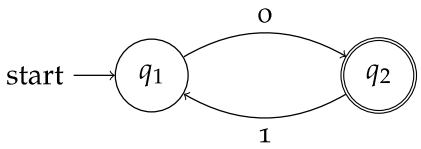


Figure 1.: Simple Finite Automaton with inputs 0/1.

In this example, the circles represent states. Arcs between states represent transitions, and they labelled with inputs 0 or 1. We have an "initial state" which is where the machine starts its execution; in this case, it is state q_1. We also have a "final state" or "accepting state" denoted by the double-line circle [Rabin and Scott (1959)]. For example, the automaton can be used to recognize specific sequences of symbols.

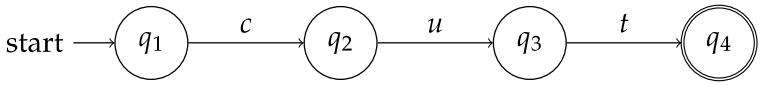


Figure 2.: Finite Automaton recognizes the word "cut".

The automaton begins in the *initial state* q_1 and waits for the input c. When it receives this input, it transits from q_1 to q_2 as shown in Fig. 2. This procedure is carried out until the state of acceptance (q_3) is reached.

2.1.1 *Alphabets and Strings*

An alphabet is a non-empty set of symbols used to represent the input set of the automaton. Usually, the symbol Σ represents an alphabet [Hopcroft et al. (2007); Aho and Ullman (1972)]. Examples include,

1. Σ = {0, 1}, which is called binary alphabet.
2. Σ = {a, ..., z, A, ...Z}, which is an alphabet that contains characters.

A *string/word* is a list over Σ. We denote the empty string by ϵ.

Example 2.1.2. *Consider the alphabets below. Σ^* is the set of all words that can be constructed, with the alphabet Σ. Below we present some alphabets and examples of words associated with them.*

1. An alphabet with some letters Σ = {a, b, c}. e.g. abb, aab, abc, a.
2. An alphabet with the numerals Σ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. e.g. 012, 987, 0, 45.
3. An alphabet combining integers and letters Σ = {0, 1, m, n}. e.g. 0m, 10n, ϵ, mn.

Computers are a particular case where the alphabet used is {0, 1}, because it is a simple way to implement this binary information via electronic components.

Example 2.1.3. *Consider two strings abba and bbab over the alphabet Σ^*={a, b}. The concatenation of these two strings is abbabbab; joining the end of string abba to the beginning of string bbab. In this operation, order is essential. The empty string ϵ has a special property in concatenation: for $r \in \Sigma^*$ $\epsilon.r = r.\epsilon = r$.*

2.2 DETERMINISTIC FINITE AUTOMATA

A deterministic finite automaton is a system with a finite set of states and a finite set of input symbols. It has an initial state, where it starts execution and may have one or more final states corresponding to the end of the execution. The transitions function is another aspect, where it receives as arguments a state and a label and returns the next state.

An automaton is deterministic if each state cannot have more than one outgoing transition with the same label. Non-determinism can arise in different ways:

1. A transition to the same state and a jump to another state with the same label.

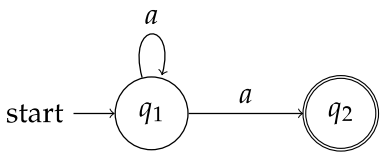


Figure 3.: Non-Deterministic Finite Automaton with 2 states

2. Two transitions, with the same origin, but with different targets and the same label. This way causes ambiguity in the system's execution.

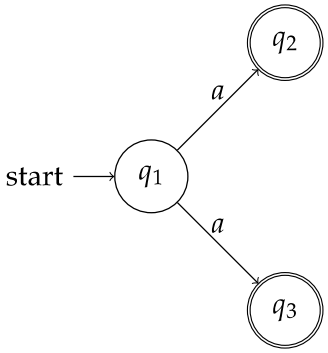


Figure 4.: Non-Deterministic Finite Automaton with 3 states

Definition 2.2.1 (Hopcroft et al. (2007))**.** A Deterministic Finite Automaton is defined as a tuple $A = (Q, \Sigma, \delta, q_0, F)$.

1. A finite set of *states*, often denoted as Q.

2. A finite set of *input symbols*, often denoted as Σ.

3. A *transition function* that takes as arguments a state and an input symbol and returns a state. The transition function will commonly be denoted as δ.

4. A *start state*, member of Q.

5. A set of *final* or *accepting* states F. The set F is a subset of Q.

A deterministic finite automaton will often be referred to by its acronym: DFA.

For an automaton to accept a sequence of symbols, first, it starts at the initial state. From there, the automaton sees the possible jumps that it can take from this position. The word is accepted if, at the end of processing the word, the automaton is in an accepting state.

Example 2.2.1. *Consider the automaton in Fig. 5. Let abb be a word. We start by analyzing the first symbol, a, with the automaton in its initial state. We see that we have a transition by a, $\delta(q_1,a) = q_1$. The next symbol in the sequence is b and the transition function in this state gives $\delta(q_2,b) = q_2$. From the function, we have that the state accepts the symbol and stays in the same state. Finally, the symbol b, which for the same reasons previously said, is also accepted. Acceptance of symbols was successful, and the automaton is in the final state, so the word is recognized.*

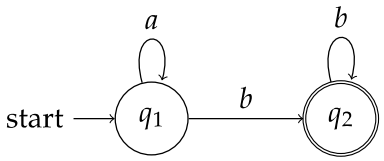


Figure 5.: Deterministic Finite Automaton

2.3 NON-DETERMINISTIC FINITE AUTOMATA

A non-deterministic finite automata (NFA) has multiple outgoing transitions labelled by the same letter and originating from a single state.

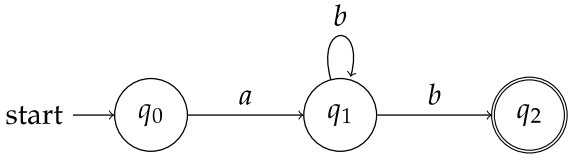


Figure 6.: Non-Deterministic Finite Automaton

Example 2.3.1. *In Fig. 6, we have a non-deterministic automaton. It recognizes sequences that start with a, and are followed by at least one b. This automaton is non-deterministic because, in state q_2, it has two transitions labeled b, $\delta(q_1,b) = \{q_1, q_2\}$.*
Fig. 7 shows how an NFA processes inputs. When a is read, the NFA can only go to the state q_1. Then, the second input, b, is read and it can go either state q_2 or q_1. If it goes to q_2 becomes stuck because there is no arc labelled b. Otherwise, if it goes through q_1, it continues the process. To recognize the third input, b, it has two option: first, goes through q_1, since it is the last input and this state is not an accepting state it becomes stuck; second, goes throw q_2 recognizing the string abb.

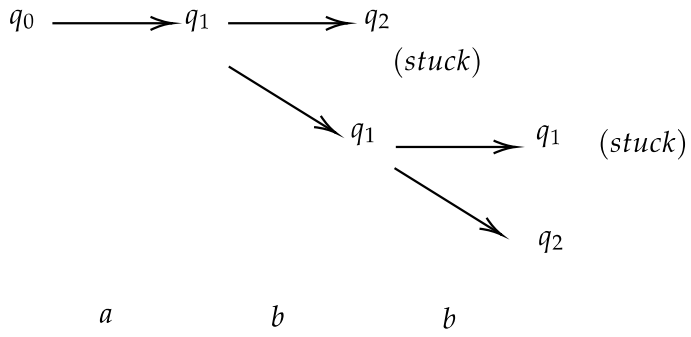


Figure 7.: The states of an NFA is during the processing the input abb

Definition 2.3.1 (Hopcroft et al. (2007))**.** A *Non-Deterministic Finite Automaton* (NFA) A = (Q, Σ, δ, q_0, F) is defined as:

1. Q is a finite set of *states*.
2. Σ is a finite set of *input symbols*.
3. q_0, a member of Q, is the start state.
4. F, a subset of Q, is the set of *final* (or accepting) states.
5. δ, the *transition function* is a function that takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q. The essential difference between NFA and DFA is in the transition function. A DFA returns an element while a NFA returns a set of states.

If an NFA has no outgoing transitions in a given state, it should be represented with the symbol $\emptyset$.

A *transition table* is a conventional tabular representation of transition functions δ, which receives two arguments and from them returns a state. The rows of the table correspond to the states, and the columns correspond to the inputs. The entry for the row corresponding to input a is the state δ(*q*,a).

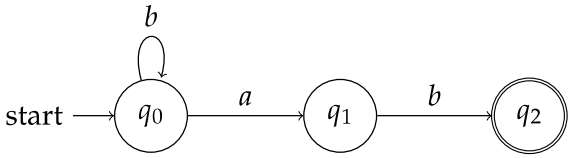


Figure 8.: Deterministic Finite Automaton

Table 1 is a tabular representation of the state transitions of the automaton in Fig. 8. The transition tables can be used to specify the transition function for both NFAs and DFAs. The difference is that each entry in the table for an NFA is a set.

	a	b
q_0	$\{q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$

Table 1.: Transition table of the automaton in Fig. 8

2.4 ω-AUTOMATA

A formal language is defined as a set of finite words over some given finite alphabet. As opposed to this, an ω-language consists of infinite words - an infinite sequence of symbols over some finite alphabet.

The automata on infinite words and automata on finite words are very similar in almost all characteristics except when it comes to word acceptance. In this regard, there are some fundamental differences. Since a run over an infinite word does not terminate, the acceptance conditions based on the condition that a set of states is visited infinitely often in an execution [Safra (1988)].

A transition table A is a tuple $\langle\ \Sigma, S, S_0, E\ \rangle$, where Σ in an input alphabet, S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, and $E \subseteq S \times S \times \Sigma$ is a set of edges [Alur and

Dill (1994)]. The automaton starts in one of the initial states, and a transition from state s to $s^{'}$ can be performed if $\langle\ s, s^{'}, a\ \rangle \in E$.

For an infinite word $\sigma = \sigma_1\ \sigma_2$ ... over the alphabet Σ, we say that

$$r : s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} s_2...$$

is a run of A over σ, s_0 is an initial state, and $\langle\ s_{i-1}, s_i, \sigma_i\ \rangle \in E$ for all $i \geq 1$. A Büchi automaton A is a transition table with set $F \subseteq S$ of accepting states. A *run* r of A over a word $\sigma \in \Sigma^{\omega}$ is an accepting run iff, some state from the set F repeats infinitely often along r, i.e. $inf(r) \cap F \neq \emptyset$, where $inf(r)$ is a set of states such that $s = s_i$ for infinitely many $i \geq 0$ [Alur and Dill (1994)]. The language that is accepted by each s $\in inf(r)$ transition table are all those in which A has an accepting run over σ.

Example 2.4.1. *Consider the automaton in Fig. 9 over the alphabet $\{a, b\}$. The automaton starts in the state q_0. A run in this automaton can be:*

$$r : q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_1 \xrightarrow{a} ...$$

The automaton accepts all the words that begin with either a or b and are followed by an infinite number of a's.

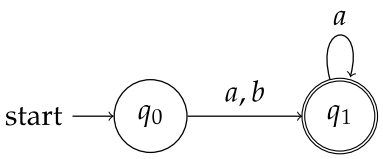


Figure 9.: Büchi Automaton recognizing $(a + b)a^{\omega}$

An ω-language is called ω-regular iff it is accepted by some Büchi Automaton.

2.5 REGULAR EXPRESSIONS

We now introduce the notion of a regular expression, which has some applications like text processing and parsing techniques.

A language consists of words whose symbols are from an alphabet and have a set of rules. A language is called regular iff it can be accepted by an automaton; also, this language can be defined by a regular expression. Therefore, not every language is regular.

Regular expressions are equivalent to finite automata because regular expressions can be created from finite automata and vice versa. In particularly, each regular expressions defines the same languages as the corresponding automaton. There are, of course, differences in their representation, automata are a more graphic form of representation whilst regular expression have a more algebraic flavour.

2.5.1 *Operations on Regular Expressions*

Regular Expressions have many associated operations which we detail next [Hopcroft et al. (2007)]:

1. The union of two languages L and M, is the set of strings that are in either L or M, or both. For example, $L = \{aa, ba, b\}$ and $M = \{b, aa, bb\}$, if we make the union of both we have $L \cup M = \{aa, ba, b, bb\}$.

2. The concatenation of languages L and M is the set of strings that can be formed by taking any string in L and concatenating it with any string in M. For example, $L = \{a, b, ab\}$ and $M = \{a, bb\}$, the concatenation of both is $LM = \{aa, abb, ba, bbb, aba, abbb\}$. We emphasise that LM and ML concatenation are not necessarily the same. The order in which they are placed is important.

3. The *closure* of a language L is denoted L^* and is defined as $L^* = \cup_{i \in N} L^i$ where L^i corresponds to the concatenation of L with itself i-times. For example, if $L = \{a, b\}$ and applying the *closure* operator $L^* = \{aa, ab, bba..\}$.

Example 2.5.1. *Let $L = \{a, ba\}$ be a language. To compute L^*, we must compute L^i for each natural number i and take the union of all these languages. Although each L^i is finite, the union of the infinite number of terms L^i is generally an infinite language. We see that $L^0 = \{\epsilon\}$, the only word of length zero is ϵ, this is true for any language.*

We also have that L^1 is equal to the language L. Considering now L^2, and we obtain {aa, baa, aba, baba}.

Regular expressions use precedence because we need to take into account the order of operations. For example, 3x5+3 is an arithmetic expression, and as we know from the arithmetic, product takes precedence over the sum. We can group as follows, (3x5)+3 continuing with the same meaning as before. We define the following precedence for operators [Hopcroft et al. (2007)]:

1. The *closure* operator has the highest precedence. It applies only to the smallest sequence of symbols to its left that is a well-formed regular expression.

2. Next in precedence comes the concatenation or "dot" operator.

3. Finally, the operator + takes precedence. Since the union is also associative, it matters little in which order consecutive unions are grouped, but we shall assume grouping from the left.

Note that even if precedence rules are implied, it can still use parenthesis.

Building Regular Expressions

To describe a regular expression, we start with its most elementary parts, such as variables (expressions) and constants (ϵ and $\emptyset$). We then construct more complex expressions by aggregating variables or constants with the operations presented in Section 2.5.1 [Hopcroft et al. (2007); Berry and Sethi (1986)]. The function *L(E)*, used below, refers to the language denoted by a regular expression *E*.

Basics of Regular expressions

1. The constants ϵ and $\emptyset$ are regular expressions, denoting the languages $\{\epsilon\}$ and $\emptyset$, respectively. That is, $L(\epsilon) = \{\epsilon\}$, and $L(\emptyset) = \emptyset$.

2. If *a* is any symbol, then *a* is a regular expression. This expression denotes the language {a}.

3. A variable, usually capitalized and italic such as *L*, is a variable representing any language.

Then

1. If E and F are regular expressions, then $E + F$ is a regular expression denoting the union of $L(E)$ and $L(F)$. That is, $L(E + F) = L(E) \cup L(F)$.

2. If E and F are regular expressions, then EF is a regular expression denoting the concatenation of $L(E)$ and $L(F)$. Thus is, $L(EF) = L(E)\ L(F)$

3. If E is a regular expression, then E^* is a regular expression, denoting the closure of $L(E)$. That is, $L(E^*) = (L(E))^*$.

4. If E is a regular expression, then (E), is also a regular expression, denoting the same language as E. Formally, $L((E)) = L(E)$.

Example 2.5.2. *Considering an alphabet $\Sigma = \{a, b\}$, we have the following example.*

$$(ab)^* + a(ab)^*$$

Let us look at each individual sub-expression. In the first one $(ab)^*$, we obtain words whose last two elements are ab. The parentheses in this expression define that the recognized language is ϵ or a finite sequence of words ab. If it had no parentheses, $a(b)^*$, this expression would recognize a different language since the precedence of the closure is greater than the concatenation.
The second is similar to the first, except that it has a symbol to the left of the expression. The language recognized, in this case, is a string that is initiated by a and whose last two letters are ab.
The languages are joined by the + operator, resulting in the union of the two languages.

2.5.2 *From DFAs to Regular Expressions*

Finite automata and regular expressions have different approaches in the way they describe the languages they accept. The two approaches accept the same class of languages.

Method of elimination of states

In the repeal we are going to introduce a method to convert a DFA into a regular expression. This method consists of eliminating states from an automaton. However, we should pay attention to some central aspects.
It begins to eliminate states that are neither accepting nor initial. When eliminating a state s, all the paths it passes through are eliminated from the automaton. Since the language remains the same, when we eliminate the states, the automaton labels are replaced by regular expressions rather than symbols. For further detail, consult Hopcroft et al. (2007).

Example 2.5.3. *Let us consider the DFA in Fig. 10 that accepts the string* ab.

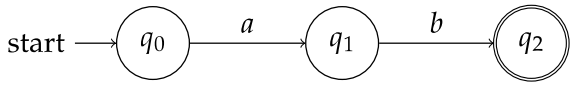


Figure 10.: An DFA accepting the string *ab*

We first remove the intermediate state (q_1) since this is neither accepting nor initial, and this will not be in the reduced automaton anyway. Then we link the remaining two states with label ab.

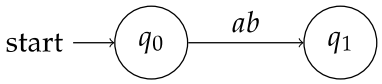


Figure 11.: Eliminating state q_1

The automaton language is the union of all path from the initial state to an accepting state.
The strategy for constructing a regular expression from a finite automaton is as follow:

1. For each accepting state q, we have applied the reduction process to produce an equivalent automaton with regular-expression labels on the arcs. Eliminate all states except q and the start state.

2. If $q \neq q_0$, then we shall be left with a two-state automaton that looks like Fig. 12. The regular expression for the accepted strings can be described in various ways.

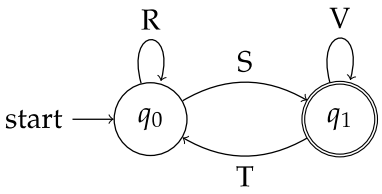


Figure 12.: Generic two-state automaton

3. If the start state is also an accepting state, we must also perform a state elimination from the original automaton that eliminates every state but the start state. When we do so, we are left with a one-state automaton that looks like Fig. 13. The regular expression denoting the string that it accepts is R^*.

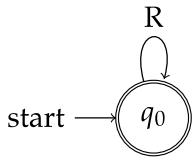


Figure 13.: Generic one state automaton

4. The desired regular expression is the sum (union) of all the expressions derived from the reduced automata for each accepting state, by rules (2) and (3).

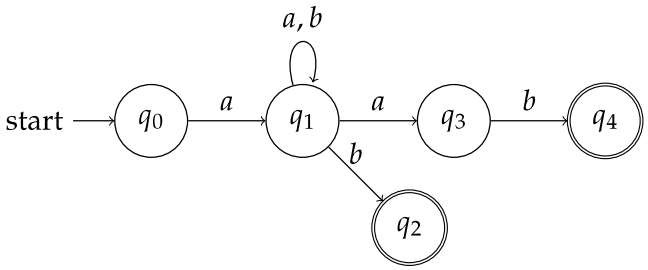


Figure 14.: An NFA that accepts strings that begin with a and end with either b or ab

Example 2.5.4. *Let us consider the NFA in Fig. 14 that accepts all strings beginning with a end either a or ab. The first step is to convert it to an automaton with regular expressions labels. Before start to eliminate state, we have to replace the label, a, b, by a regular expression **a+b**.*

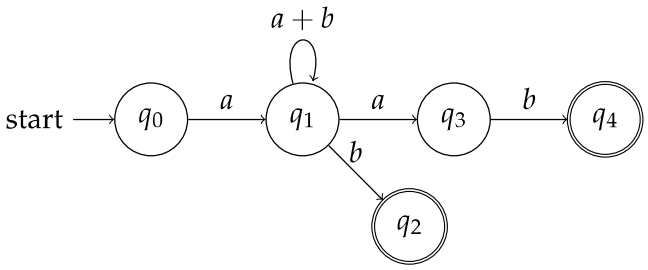


Figure 15.: The automaton of Fig. 14 with different labels

Let us first eliminate state q_1, since this state is neither accepting nor initial. State q_1 has one predecessor, q_0, and two successor, q_2 and q_3. So we will replace the deleted state and its links for two new arrows with new labels from the qo state. The first path with the arc, $q_0 \rightarrow q_3$, consists of the concatenation of all the labels of this path. These labels are respectively a,$(a + b)^*$ and a. Noting that as the q_1 state had a self-loop a+b, the equivalent in the regular expression is to place parentheses

around the expression with a Closure. Thus, the final expression is $a(a+b)^*b$.
Similarly, the second path corresponds to the arc $q_0 \to q_2$, *and the expression is* $a(a+b)^*a$.

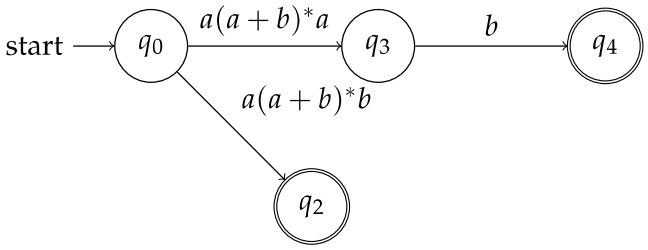


Figure 16.: Eliminating state q_1

To eliminate state q_3, *the mechanics are similar to those we performed above to eliminate the state* q_1. *And, the resulting shown in Fig. 17.*

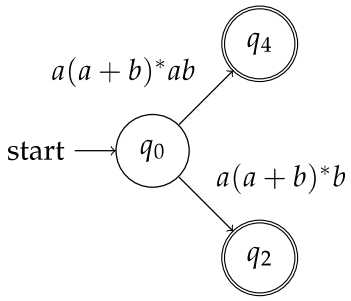


Figure 17.: Eliminating state q_3

Now, we have two final states, and there are no successors. So one of these states is going to be eliminated, and the two expressions will unite.

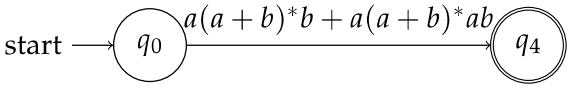


Figure 18.: A two-states automaton with states q_0 and q_4

The two-state automaton is equivalent to that shown initially and represented in Fig. 18, the regular expression is:

$$a(a+b)^*b + a(a+b)^*ab$$

Brzozowski Algebraic Method

The Brzozowski method is another approach to generating regular expressions. This method is based on regular expressions, where each state has an associated regular expression, and then we solve the system concerning the starting state. Later, we will use this method as a guiding tool to develop our algorithm.

An automaton is denoted by M, and each state of it is denominated by q_i. A characteristic equation is an equation that characterizes a state of an automaton. To build the characteristic equations, we must take into account the following. Each term can be constructed as follows: for a transition a from q_i and q_j, the therm is aR_j, where a is a label and R_j the equation from state q_j. The symbol λ, used next, marks the final state [Neumann (2005)]. This leads to a system of equations:

$$\begin{aligned} R_1 &= a_1R_1 + a_2R_2 + ... \\ R_2 &= a_1R_1 + a_2R_2 + ... \\ R_3 &= a_1R_1 + a_2R_2 + ... + \lambda \\ \vdots &= \vdots \\ R_m &= a_1R_1 + a_2R_2 + ... + \lambda \end{aligned}$$

We now need to solve the system of regular expressions. We isolate R_i on the left side and successively substitute the R_i and R_j on the right side. We begin with those marked with a final state. We solve the system for the regular expression associated with the initial state. If the expression that will be substituted is the expression itself, the substitution is not performed. When this happens, it means that the q_i state has a self-loop. It is solved using Arden's theorem [Neumann (2005)], which is as follows:

Given an equation of the form $X = AX + B$ where $A \notin \lambda$, the equation has a unique solution $X = A^*B$.

When we find this situation, we use Arden's theorem to isolate on the left the occurrences of this expression on the right.

Example 2.5.5. *Consider the following automaton, which is the same one as in Fig. 14. We will use this example to demonstrate the Brzozowski method.*

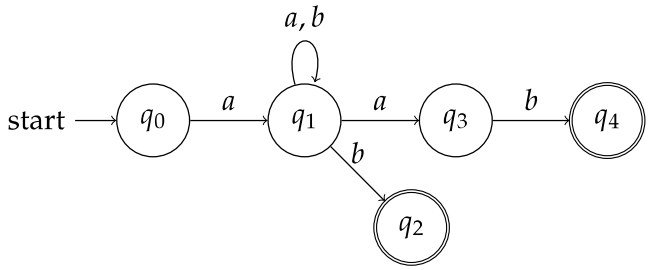


Figure 19.: NFA which accepts strings that begin with a and end with either b or ab

The system of regular expressions is as follows:

$$
\begin{aligned}
R_0 &= aR_1 \\
R_1 &= aR_1 + bR_1 + aR_3 + bR_2 \\
R_2 &= \epsilon \\
R_3 &= bR_4 + \lambda \\
R_4 &= \epsilon
\end{aligned}
$$

We first solve R_3, by replacing R_4 with the corresponding expression.

$$
\begin{aligned}
R_3 &= b(\epsilon) + \lambda \\
&= b + \lambda \\
&= b
\end{aligned}
$$

Then we substitute in the expression R_1 the expressions R_3, with substitution made previously, and R_2. After that, we use Arden's theorem to deal with the expression R_1 since it appears on both sides of the expression.

$$\begin{aligned} R_1 &= aR_1 + bR_1 + aR_3 + bR_2 \\ &= aR_1 + bR_1 + a(b) + b(\epsilon) \\ &= (a+b)R_1 + (ab+b) \\ &= (a+b)^*(ab+b) \end{aligned}$$

Finally, we substitute into R_0 the expression R_1, maintaining the alterations made earlier.

$$\begin{aligned} R_0 &= aR_1 \\ &= a(a+b)^*(ab+b) \end{aligned}$$

Thus, the regular expression for the automaton in Fig. 19 is:

$$a(a+b)^*(ab+b)$$

3

WHEN REAL-TIME COMES INTO SCENE

3.1 SUMMARY

As we previously defined in Section 2 regular expressions and their equivalence. Now we introduce a temporal dimension to these systems. For this, we need models that allow us to include such a dimension based a suitable notion of time.

Classic automata do not harbour a notion of time. In a real-time system it is vital a model which has the concept of time incorporated in it, and capable of handling an infinite sequence of symbols. So, it is necessary to include a notion of time.

3.2 A THEORY OF TIMED AUTOMATA

A timed automaton is essentially a finite automaton with real-time variables. These automata incorporate a timed model (model with defined characteristics for handling the time domain) [Bengtsson and Yi (2003)]. The system is based on clock variables, which are initialized to 0 when the automaton starts. All clocks increments at the same rate, and clock reset can occur only during transitions.

3.2.1 *Timed Languages*

Timed words are infinite sequences in which a real-valued time of occurrence is associated with each symbol [Alur and Dill (1994)]. In this system, time does not decrease, and the time domain adopt the real line, **R**.

Definition 3.2.1 (Alur and Dill (1994))**.** A time sequence $\tau = \tau_1\ \tau_2\ ...$ is an infinite sequence of time values $\tau_i \in \mathbf{R}$ with $\tau_i > 0$, satisfying the following constraints:

1. Monotonicity: τ increases in a strict monotonic way.
2. Progress: For every $t \in \mathbf{R}$, there is some $i \geq 1$ such that $\tau_i > t$.

A timed word consist of pair (σ, τ) where $\sigma = \sigma_1\sigma_2 ...$ is an infinite word over Σ and τ is a time sequence. A *timed language* over Σ is a set of timed words over Σ.

An automaton sees a timed word (σ, τ) as input and interprets it as follows: receives a symbol σ_i, meaning that an event has occurred, and the time it happened was τ_i.

Example 3.2.1. *Let the alphabet be $\{a, b, c\}$. Consider a timed language L_1 whose timed words are those that in the first two time units only have b's.*

$$L_1 = \{(\sigma, \tau) | \forall_i.(\tau_i < 2) \rightarrow (\sigma_i = b)\ \}$$

An hypothetical run for this language can be:

$$r : s_1 \xrightarrow[1]{b} s_2 \xrightarrow[1.5]{b} s_3 \xrightarrow[2.5]{c} s_4 \xrightarrow[3]{a} ...$$

Definition 3.2.2 (Alur and Dill (1994))**.** For a timed language L over Σ, *Untime(L)* is the ω-language consisting of $\sigma \in \Sigma^\omega$ such that $(\sigma, \tau) \in L$ for some time sequence τ.

Example 3.2.2. *Consider a run r over the language* $\mathrm{L} = (a + b)^\omega$*:*

$$r : s_1 \xrightarrow[1]{b} s_2 \xrightarrow[1.7]{a} s_3 \xrightarrow[2.2]{b} s_4 \xrightarrow[3]{b} s_4 \xrightarrow[3]{b} s_5 \xrightarrow[3.4]{b} s_6 \xrightarrow[4]{b} ...$$

The untimed of this particular case can be seen as all single symbols that occurred in the execution without the time element.

$$r : s_1 \xrightarrow{b} s_2 \xrightarrow{a} s_3 \xrightarrow{b} s_4 \xrightarrow{b} s_4 \xrightarrow{b} s_5 \xrightarrow{b} s_6 \xrightarrow{b} ...$$

3.2.2 *Timed transition tables*

Previously, when a state transition occurs, it only depended on the input symbol read. Now we want this transition also to depend upon the time of the input symbol, relatively to the previous transition [Alur and Dill (1994)]; so that it will be able to deal with timed words, and eventually, change the dynamics of how the words are accepted. The automata choose the next state depending only on the input symbol read. Now, for timed transition tables, the transition to the next state depends on the input symbol previously read, and is subjected to timed constraints [Alur and Dill (1994)]. Put in another way, the choice of the following state depends upon the time of the input symbol relative to the time that the previous read symbols. In any transition, the clocks can set to o (reset) simultaneously. Each transition is associated with a clock, and these clocks have restrictions associated. Therefore, transitions are only allowed if the current time value satisfies the clock constraints.

Definition 3.2.3 (Alur and Dill (1994))**.** A *timed transition table* A is a tuple $\langle\ \Sigma, S, S_0, C, E\ \rangle$, where:

1. Σ is a finite alphabet.
2. S is a finite set of states.
3. $S_0 \subseteq S$ is a set of start states;
4. C is a finite set of clocks.
5. $E \subseteq S \times S \times \Sigma \times 2^C \times \phi(C)$ is the set of transitions. An edge $\langle\ s, s', a, \lambda, \delta\ \rangle$ represents a transition from state s to state s' on input symbol a. The set $\lambda \subseteq C$ gives the clocks to be reset on this transition, and δ is a clock constraints over C. Clock constraint compares a clock value with a time constant, and allows Boolean combinations of simple constraints.

Definition 3.2.4 (Alur and Dill (1994))**.** A run r, denoted by $(\overline{s}, \overline{v})$, of a timed transition table $\langle\ \Sigma, S, S_0, C, E\ \rangle$ over a timed word (σ,τ) is an infinite sequence of the form

$$r : \langle\ s_0, v_0\ \rangle \xrightarrow[\tau_1]{\sigma_1} \langle\ s_1, v_1\ \rangle \xrightarrow[\tau_2]{\sigma_2} \langle\ s_2, v_2\ \rangle \xrightarrow[\tau_3]{\sigma_3} \ldots$$

with $s_i \in S$ and $v_i \in [C \to \mathbf{R}]$, for all $i \geq 0$, satisfying the following requirements:

1. *Initiation*: $s_0 \in S_0$, and $v_0(x) = 0$ for all $x \in C$.
2. *Consecution*: for all $i \geq 1$, there is an edge in E of the form $\langle\ s_{i-1}, s_i, \sigma_i, \lambda_i, \delta_i\ \rangle$ such that $(v_{i-1}+\tau_i - \tau_{i-1})$ satisfies δ_i and v_i equals $[\lambda_i \to 0](v_{i-1}+\tau_i - \tau_{i-1})$, this means that

set of clocks λ_i will be reseted and $(v_{i-1}+\tau_i - \tau_{i-1})$ ensures the progression of time on the clocks.

The set *inf(r)* consists of those states $s \in S$ such that $s = s_i$ for infinitely many $i \geq 0$.

Example 3.2.3. *Consider the following* timed word*:*

$$(a,1) \rightarrow (b,1.5) \rightarrow (d,2) \rightarrow (a,4) \ldots$$

A clock interpretation is represented by [x].

$$\mathrm{r} : \langle\, q_0, [0] \,\rangle \xrightarrow[1]{a} \langle\, q_1, [0] \,\rangle \xrightarrow[1.5]{b} \langle\, q_2, [0.5] \,\rangle \xrightarrow[2]{d} \langle\, q_2, [1] \,\rangle \xrightarrow[4]{a} \langle\, q_2, [0] \,\rangle \ldots$$

Within the brackets, we have two parameters, the first one is the current state, and the second provides the current values for the available clocks. On top of the arrows, we place the symbols of the transition. And, at the bottom refers to the overall clock of the system.

3.2.3 *Transition tables with timing constraints*

Example 3.2.4. *Consider Fig. 20. The start state is q_0. In this timed transition table, there is a single clock. Edges have expressions such as $x := 0$, which puts the clocks to o. Expressions such as $(x < 2)?$ are constraints of transition.*

The automaton starts in the state q_0 and moves to state q_1, reading input symbol a, if the value of clock x is higher than 3. In state q_1, it can stop right here, since this is an accepting state, or it can go to state q_2, resetting the clock x and reading the symbol b. The last state is q_2, which can perform a transition back to q_1, if the value of clock x is lower than 2, and reads symbol a. The language L_1 represents the automaton in Fig.20

$$L_1 = \{(a(ba)^\omega, \tau) | \tau_0 > 3 \wedge \forall_i.(\tau_{2i} < \tau_{2i-1} + 2)\}$$

The language L_{01} expresses the constraints of the edge $q_0 \rightarrow q_1$, and L_{12} express the transitions between q_1 and q_2.

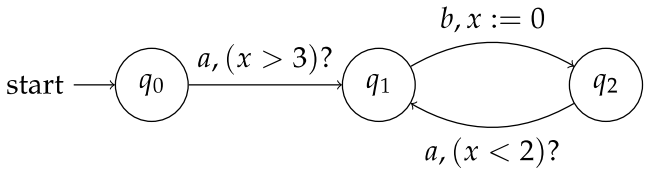


Figure 20.: Timed transition table

3.2.4 *Clock constraints*

Edges, in addition to carrying symbols, also include constraints concerning clocks. These constraints limit the possible transitions that an automaton can take. We need to define formally clock constraints. A transition may be taken only if the current values of the clocks satisfy the associated constraints [Alur and Dill (1994),Alur et al. (1993)]. We can build more complex ones from a simple constraint, but they all have to be Boolean combinations. As previously said in this chapter, time constants have to be non-negative.

Definition 3.2.5 (Alur and Dill (1994))**.** For a set X of clock variables, the set $\Phi(X)$ of *clock constraints* δ is defined by:

$$\delta := x \leq c \mid c \leq x \mid \neg\, \delta \mid \delta_1 \wedge \delta_2$$

where x is a clock in X and c is a constant in Q^+.

From this grammar we can build constraints such as **true**, $(x = c)$, $x \in [2, 5)$.

3.2.5 *Timed regular language*

Timed regular languages can be designated as an automaton; all the time constraints assigned to it must comply with the following rule, each time constraint, the time interval between two successive pairs, a and b, cannot increase. Put another way, the class of regular timed languages are all of those who are accepted by a timed Büchi automaton.

Definition 3.2.6 (Alur and Dill (1994))**.** A *timed Büchi automaton* is a tuple $\langle \Sigma, S, S_0, C, E, F \rangle$, where $\langle \Sigma, S, S_0, C, E \rangle$ is a timed transition table, and $F \subseteq S$ is a set of *accepting* states.

A run $r = (\overline{s}, \overline{v})$ of the automaton over a timed word(σ,τ) is called an *accepting run* iff $inf(r) \cap F = \emptyset$, in other words, the set of states which are visited infinitely, needs to contain at least one of the accepting states.

For a *timed Büchi automaton* A, the language L(A) of timed words it accepts is defined as the set $\{(\sigma,\tau)|$ has an accepting run over $(\sigma,\tau)\}$.

Example 3.2.5. *Consider the timed Büchi automaton in Fig. 21 over the alphabet $\{a, b\}$.*

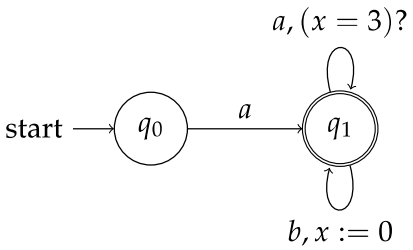


Figure 21.: Timed Büchi Automaton

The start state is q_0, the accepting state is q_1, and there is a single clock x. The automaton starts in the state q_0 and jumps to state q_2 without constraints. While in state q_1, there are two self-loops. The first one is $a, (x = 3)?$; this label expresses a possible jump if the value of the clock x is equal to 3. The second one, $x := 0$, without restrictions, the clock x is reset and reads b in the process in this transition.

4

TIMED REGULAR EXPRESSIONS

4.1 INTRODUCTION

In chapters 2 and 3, we studied classical automata and timed variants; in particular timed automata and timed regular expressions. Their essential concepts have been addressed, and these will be important to introduce timed regular expressions, which will take a similar role to that regular expressions have for classical automata. Note that our notion of a timed regular expression is different from the one briefly described in the previous section: the latter aims at encoding all possible executions of a timed automaton whilst ours corresponds to an algebraic description of its behaviour and has therefore a more syntactic behaviour. It is reflected, for instance in the fact that our notion of timed regular expression includes Boolean formulas describing clock constraints whilst the previous notion encodes all possible executions satisfying these clock constraints. Among the methods mentioned in Section 2.5.2, we chose the Brzozowski method because it follows a more algebraic approach which is easier to implement.

We begin with a similar problem to give a slight glance at what we are trying to achieve with the algorithm that we developed, which the formal definition is in Section 4.3. Later, we get into more in detail about it. Imagine a simple arithmetic expression with some variables. These variables can be simple arithmetic expressions with or without variables in them.

Example 4.1.1. *For example, consider the expressions below:*

$$w = 2 + x + 5 * x$$
$$x = 10 * y$$
$$y = 62$$

These expressions have some variables that can be substituted. We start with a simple substitution, namely the substitution of y by 62.

$$w = 2 + x + 5 * x$$
$$x = 10 * 62$$
$$y = 62$$

With updated expressions, we proceed by substituting x:

$$w = 2 + (10 * (62)) + 5 * (10 * (62))$$
$$x = 10 * (62)$$
$$y = 62$$

This example is straightforward. Problems arise when the variable that we are trying to substitute appears in both the left and right-hand side of the equation, for example, $x = 10 * x$. In these cases, we need to develop a way of not getting into an infinite loop during the substitution process. To deal with this, we look at Silva (2010). Silva's work introduces a feature that regarded as a variable binder μ. At the beginning of the expression, we put this binder.

We consider an expression *locked* if it is under a binder (μ). When an expression is under a binder, the substitution for the binder is *locked*. For example, the binder μ_x blocks any substitution under the variable x.

Example 4.1.2. *Consider the following example. These expressions will be locked in the beginning under the expression they are in.*

$$w = \mu_w(x + 5 * w)$$
$$x = \mu_x(10 * x)$$

There is only one free, variable x in the expression w

$$w = \mu_w((\mu_x(10 * x)) + 5 * w)$$
$$x = \mu_x(10 * x)$$

Since we are dealing with timed automata, we need to add one new thing: The notion of a *guard*. This new label comprises an event, a guard and a reset, displayed in Fig. 22, replacing the standard case of classical automata where only labels are involved at transitions. Resets

correspond to a programming language, but for mathematical purposes, it is easier to treat as a Boolean expression. The introduction of more elements to regular expressions has a cost. Readability becomes more complex and consequently more difficult to understand.

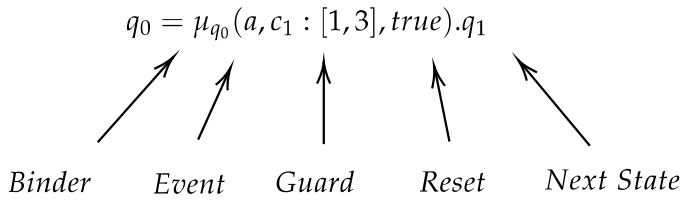


Figure 22.: Description of all components of a Timed Regular Expression

Example 4.1.3. *Consider the following timed regular expression.*

$$q_0 = \mu_{q_0}((a, x < 3, true).q_0 + (b, true, x := 0).q_0)$$

Figure 23.: A Timed Regular Expression

We see that the expression q_0 starts with a binder to this same expression, this way we prevent infinite loops. Then follows the expression that is under this binder $((a, x < 3, true).q_0 + (b, x = 3, x := 0).q_0)$. Within this one we have two transitions: $(a, x < 3, true).q_0$ and $(b, x = 3, x := 0).q_0$. The first one has an event a*, a clock restriction $x < 3$, a clock reset true and a next state q_1. If the clock transition ($x < 3$) is satisfied, then transits to the state q_0 ($a, x < 3, true$ and $b, x = 3, x := 0$), and, since we have a statement true we have no clock to perform reset. Otherwise, it remains in the current state. The second one has an event b, and does not have any clock restriction, a clock reset $x := 0$ and a next state q_0. So, this transition can be made at the time of its request, accompanied by a clock $x := 0$.*

4.2 From timed automata to timed regular expressions (a sketch)

In this section, we sketch a method for converting an automaton into a regular expression (the following section presents an implementation of this method). The automaton in Fig. 24 is given as an input to our algorithm, shown in the next section. Below we have an automaton to demonstrate how the algorithm proceeds when it receives an input.

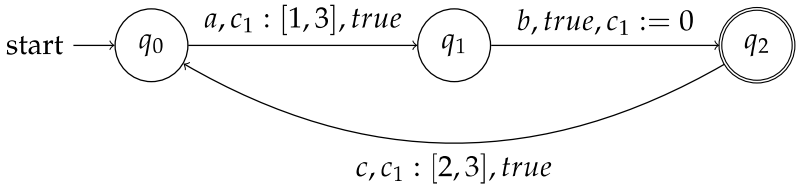


Figure 24.: A timed automaton with three state

The automaton, yields the following important data:

$$\begin{aligned}
Intial\ state &- q0 \\
Set\ of\ states &- q0, q1, q2 \\
Set\ of\ clocks &- c1 \\
Set\ of\ pairs &- no(does\ not\ have\ any\ set\ of\ pairs,\ so\ we\ put\ the\ statement\ no) \\
Set\ of\ transitions &- (q0, (a, c1 : [1,2], true), q1), (q1, (b, true, c1 := 0), q2), (q2, (c, c1 : [2,3], true), q0)
\end{aligned}$$

In Section 5.1.1, we explain in more detail the parsing. The information that we give as an input is the one mentioned above. Now, with this information, the algorithm produces the expressions.

$$\begin{aligned}
q_0 &= \mu_{q_0}(a, c_1 : [1,3], true).q_1 \\
q_1 &= \mu_{q_1}(b, true, c_1 := 0).q_2 \\
q_2 &= \mu_{q_2}(c, c_1 : [2,3], true).q_0
\end{aligned}$$

It has three expressions that correspond to the three states of the automaton. These expressions are inserted into a list and begin in the first expression (q_0). The algorithm has some variables that are used to help in the generation of regular expressions. The first variable is n, an integer that indicates which of the expressions is in and is also a stop condition. Then we have a list of FS free states; i.e. the set of states that can be replaced with expressions. This list is updated on each substitution made to the expression.

We start with the expression (a)(q_0); we see that it has an free state FS=$\{q_1\}$. Given this, we search for the expression q_1 in the list. After finding it repiting in the current expression, originating the expression (b)(q_0). With this substitution, we move on to the following set of

expressions named (b). In expression (b)(q_0) has a free state FS= $\{q_2\}$. Following the same procedure as in the previous expression, finds the expression q_2 and replaces it. Finally, we come to the set of expressions (c). Since it has no free states FS=$\emptyset$, which means that the expression is already in the final form; therefore, the n is incremented and passes to the next expression. As long as there are free states in a given expression, n does not increment. This process continues until we reach the last expression of the list and FS becomes empty.

(a)

$$
\begin{aligned}
q_0 &= \mu_{q_0}(a, c_1 : [1,3], true).q_1 \\
n &= 0, FS = [q_1] \\
q_1 &= \mu_{q_1}(b, true, c_1 := 0).q_2 \\
q_2 &= \mu_{q_2}(c, c_1 : [2,3], true).q_0
\end{aligned}
$$

(b)

$$
\begin{aligned}
q_0 &= \mu_{q_0}(a, c_1 : [1,3], true).(\mu_{q_1}(b, true, c_1 := 0).q_2) \\
n &= 0, FS = [q_2] \\
q_1 &= \mu_{q_1}(b, true, c_1 := 0).q_2 \\
q_2 &= \mu_{q_2}(c, c_1 : [2,3], true).q_0
\end{aligned}
$$

(c)

$$
\begin{aligned}
q_0 &= \mu_{q_0}(a, c_1 : [1,3], true).(\mu_{q_1}(b, true, c_1 := 0).(\mu_{q_2}(c, c_1 : [2,3], true).q_0)) \\
n &= 0, FS = [] \\
q_1 &= \mu_{q_1}(b, true, c_1 := 0).q_2 \\
q_2 &= \mu_{q_2}(c, c_1 : [2,3], true).q_0
\end{aligned}
$$

(d)

$$q_0 = \mu_{q_0}(a, c_1 : [1,3], true).(\mu_{q_1}(b, true, c_1 := 1).(\mu_{q_2}(c, c_1 : [2,3], true).q_0))$$
$$q_1 = \mu_{q_1}(b, true, c_1 := 0).q_2$$
$$n = 1, FS = [q_2]$$
$$q_2 = \mu_{q_2}(c, c_1 : [2,3], true).q_0$$

(e)

$$q_0 = \mu_{q_0}(a, c_1 : [1,3], true).(\mu_{q_1}(b, true, c_1 := 0).(\mu_{q_2}(c, c_1 : [2,3], true).q_0))$$
$$q_1 = \mu_{q_1}(b, c_1 := 1, true).(\mu_{q_2}(c, c_1 : [2,3], true).q_0)$$
$$n = 1, FS = [q_0]$$
$$q_2 = \mu_{q_2}(c, c_1 : [2,3], true).q_0$$

(f)

$$q_0 = \mu_{q_0}(a, c_1 : [1,3], true).(\mu_{q_1}(b, true, c_1 := 0).(\mu_{q_2}(c, c_1 : [2,3], true).q_0))$$
$$q_1 = \mu_{q_1}(b, true, c_1 := 0).(\mu_{q_2}(c, c_1 : [2,3], true).(\mu_{q_0}(a, c_1 : [1,3], true).(\mu_{q_1}(b, true, c_1 := 0)$$
$$.(\mu_{q_2}(c, c_1 : [2,3], true).q_0))))$$
$$q_2 = \mu_{q_2}(c, c_1 : [2,3], true).q_0$$
$$n = 2, FS = [q_0]$$

(g)

$$q_0 = \mu_{q_0}(a, c_1 : [1,3], true).(\mu_{q_1}(b, c_1 := 1, true).(\mu_{q_2}(c, c_1 : [2,3], true).q_0))$$
$$q_1 = \mu_{q_1}(b, c_1 := 1, true).(\mu_{q_2}(c, c_1 : [2,3], true).(\mu_{q_0}(a, c_1 : [1,3], true).(\mu_{q_1}(b, true, c_1 := 0).$$
$$(\mu_{q_2}(c, c_1 : [2,3], true).q_0))))$$
$$q_2 = \mu_{q_2}(c, c_1 : [2,3], true).(\mu_{q_0}(a, c_1 : [1,3], true).(\mu_{q_1}(b, true, c_1 := 0)$$
$$.(\mu_{q_2}(c, c_1 : [2,3], true).q_0)))$$
$$n = 2, FS = []$$

4.3 THE FORMAL DEFINITION OF TIMED REGULAR EXPRESSION & EXAMPLES OF THE CONVERSION METHOD

We begin by defining how the tool developed and some differences between our approach and the theory presented in the previous chapters. First, we take a look at the structure of regular expressions. As we saw above, *timed regular expression* require more features, such as *resets*, *clock constraint*, and *events*. So all these components are placed in the same place where the label was. This way, the length of an expression increases drastically.

The automata theory community often utilizes a set of initial states and takes advantage of the notion of an execution run. In contrast, our approach does not depend on the execution run and assumes the existence of a unique initial state. As future work, we would like to consider multiple initial states. Recall the definition of timed automata presented in the previous chapter. In Section3.2.5, we have a definition of a *timed Büchi automata*, and, since our approach is slightly different, we create our definition of a timed automaton.

Definition 4.3.1. A timed automaton consists of:

1. A finite set of *states*, often denoted by Q;
2. A finite set of of *clocks*, denoted by C;
3. A finite set of *events*, denoted by E;
4. A *start state*, one of the states in Q;
5. A *transition relation*, whose elements take the form $(q, \beta, \phi, \rho, q')$ where q and q' are states, β is an event, $\beta \in E$, $\rho \subseteq C$, ϕ is clock constraint which is a combination of formulae of the form $c \in C$ for some clock c and some integer-bounded interval I. This is further detailed later on. Note that clock resets are shown in a explicit form, $x := 0$, but formally should be interpreted as a set of clocks;

A *clock valuation* is a function $\nu : C \to \mathbf{R}^+ \cup \{0\}$. For a clock valuation ν and a set $\rho \subseteq C$ we put for any clock variable $c \in C$:

$$Reset_\rho\ \nu(c) = \begin{cases} 0 & if\ c \in \rho \\ \nu(c) & if\ c \notin \rho \end{cases}$$

Definition 4.3.2. Let C be a set of clock variables. The set of *clock constraints* ϕ is defined by:

$$\phi ::= c \leq a \mid a \leq c \mid c = a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$$

where $c \in C$, and a is a constant in $\mathbf{R}^+$.

Note that the grammar above allows for expressions such as true, $a \in [2, 5]$.

Definition 4.3.3. Let C be a set of clocks and let X be a set of variables. The set Exp of *expressions* is given by the following syntax. For $c \in C$, $x \in X$:

$$v ::= true \mid x \mid v + v \mid \mu x.\gamma$$

where $\gamma \in Exp_g$, the set of guarded expressions, which is given by

$$\gamma ::= true \mid \gamma \wedge c \mid \gamma + \gamma \mid \mu x.\gamma$$

The set Exp contains closed timed expressions. An expression is closed if a variable x always occurs under the scope of the binder $\mu v.\gamma$.

Next we focus on the conversion method between timed automata and regular expressions, and also on the respective tool which is the main contribution of this dissertation. The main goal of this tool is to convert an automaton into a regular expression. The tool's mode of operation is as follows: first you must click the executable, then a file manager will appear; this file manager is used to select the document that we want to use as input for the tool. After selecting the file, double-click on it or click enter, and the conversion will appear.

Now, we will present examples of timed automata. We will compare the results between the expressions we should get and those generated by our program (implementation details in the next section). We start with simple examples at first and progressively increase the degree of difficulty. Later on, we will show some case studies of a real-time system. Some images in these examples do not have a good resolution (those generated by the algorithm); this is because the Gloss library does not let control the resolution of generated images. As future work, we would like to consider other Haskell Library to provide a higher resolution output.

The structure and decisions of the timed regular expression are presented in more detail in Section 5.1.2.

Example 4.3.1. *The following is an example of a timed automaton from the article Alur and Dill (1994). The automaton has three states. The automaton begin in the state q_0, eventually progresses and jumps to the state, q_1, as long it satisfies the guard's transitions, it can jump infinitely between these three states.*

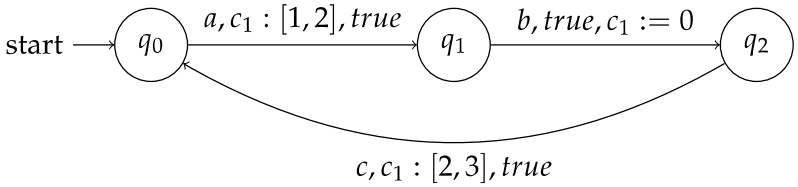


Figure 25.: A timed automaton with three states

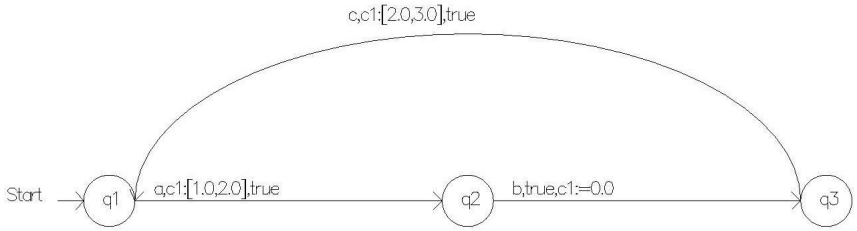


Figure 26.: A timed automaton with three state generated through our algorithm

The two automatons are the same, and we can see that they have the same transitions with the same labels. The Fig.26 is the automaton generated, and, as we can see it is in low resolution, but it can be perceived well.

$$
\begin{aligned}
q_0 &= \mu_{q_0}((a, c_1 : [1,2], true).X) \quad Where \\
X &= \mu_{q_1}((b, true, c_1 := 0).Y) \\
Y &= \mu_{q_2}((c, c_1 : [2,3], true).q_0)
\end{aligned}
$$

Figure 27.: A timed regular expression

```
Mu q1 ((a,c1:[1.0,2.0],true).X) Where
X = Mu q2 ((b,true,c1:=0.0).Y)
Y = Mu q3 ((c,c1:[2.0,3.0],true).q1)
```

Figure 28.: A timed regular expression generated through our algorithm

The regular expressions in Fig. 27 and Fig. 28 are very similar as well. Both, the automaton and regular expression generated have a decimal part in the numbers presented. It occurs because the function we use to parse numbers converts any possible number into a float.

Example 4.3.2. *The following example, again from the article Alur and Dill (1994), describes a single state timed automaton. It has three possible set events, a, b and c. The first two events, a and b, have the same guard, $x < 3$, and reset label, true. The other event, c, resets the clock, $x := 0$.*

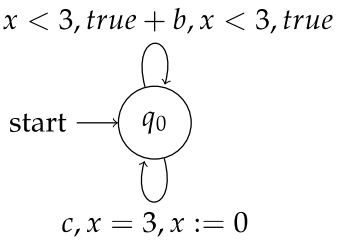


Figure 29.: A timed automaton with a single state

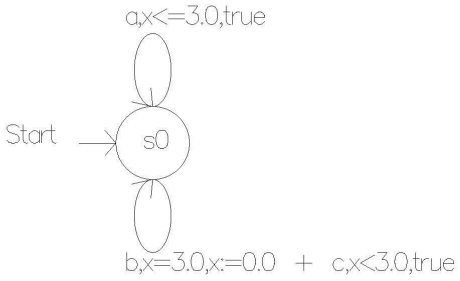


Figure 30.: A timed automaton with a single state generated through our algorithm

$$s_0 = \mu_{s_0}((a, x < 3, true).s_0 + (c, x < 3, true).s_0 + (b, x = 3, x := 0).s_0)$$

Figure 31.: A timed regular expression

Mu s0 ((a,x<=3.0,true).s0 + (b,x=3.0,x:=0.0).s0 + (c,x<3.0,true).s0)

Figure 32.: A timed regular expression generated through our algorithm

Example 4.3.3. *Consider the automaton in Fig. 33. The automaton starts in state s_0 and can either make an unspecified number of jumps between the two states, s_0 and s_1 or jump straight to the accepting state s_2.*

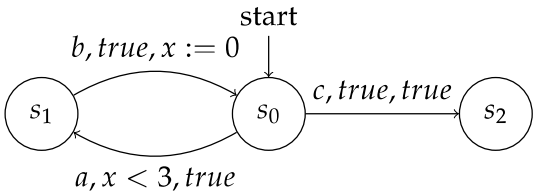


Figure 33.: A timed automaton with three states

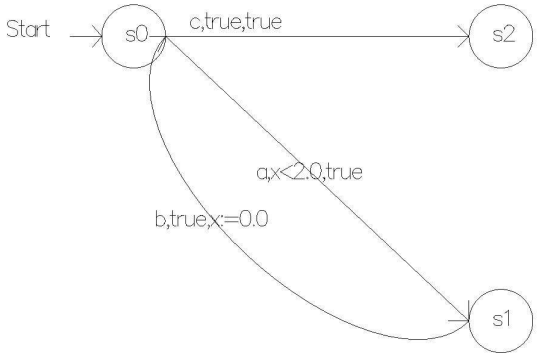


Figure 34.: A timed automaton with three state generated through the algorithm

It can be perceived that both have the same form, and they agree with the transitions guards.

$$
\begin{aligned}
&s_0 = \mu_{s_0}((c, true, true).Y + (a, x < 3, true).s_1) \quad Where \\
&X = \mu_{s_1}((b, true, x := 0).s_0) \\
&Y = \mu_{s_2}(true)
\end{aligned}
$$

Figure 35.: A timed regular expression

```
Mu s0 ((a,x<2.0,true).X + (c,true,true).Y /\ (Cl x)) Where
X = Mu s1 ((b,true,x:=0.0).s0)
Y = Mu s2 (True)
```

Figure 36.: A timed regular expressions generated through our algorithm

The state s_1 does not have any transitions from it, which is represented in the expression by the statement true in the sub-expression Y.

Example 4.3.4. *Consider the following automaton, also from the article Alur and Dill (1994). All the states - s_0, s_1, s_2 - can be visited infinitely. The initial state, s_0, can take two paths. The first one is the outgoing arrow to state s_1 and, consequently, jump back to the start. The second path, throw another outgoing arrow to state s_2.*

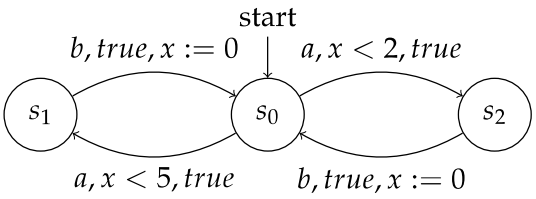


Figure 37.: A timed automaton

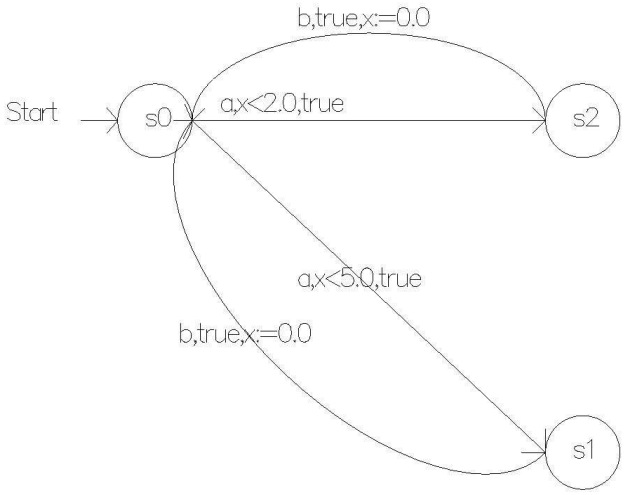


Figure 38.: A timed automaton generated through our algorithm

In Fig.38, the state s_0 is a little difficult to perceive the incoming arrows due to the lack of resolution. Putting this aspect aside, we see that they look is a little different, but have the jumps to the same states and with the same labels. The automaton generated always begins with the start state because it acts as a reference to the overall drawing.

$$s_0 = \mu_{s_0}((a, x < 5, true).X + (a, x < 2, true).Y) \quad where$$
$$X = \mu_{s_1}((b, true, x := 0).s_0)$$
$$Y = \mu_{s_2}((b, true, x := 0).s_0)$$

Figure 39.: A timed regular expression

```
Mu s0 ((a,x<5.0,true).X + (a,x<2.0,true).Y) Where
X = Mu s1 ((b,true,x:=0.0).s0)
Y = Mu s2 ((b,true,x:=0.0).s0)
```

Figure 40.: Timed automaton regular generated through our algorithm

Example 4.3.5. *Consider the timed automaton in Fig. 41, from the article Alur and Madhusudan (2004). The automaton transits progressively along the states -s_0, s_1,s_2- until it reaches the state s_4, which has two transitions that can be infinitely switched.*

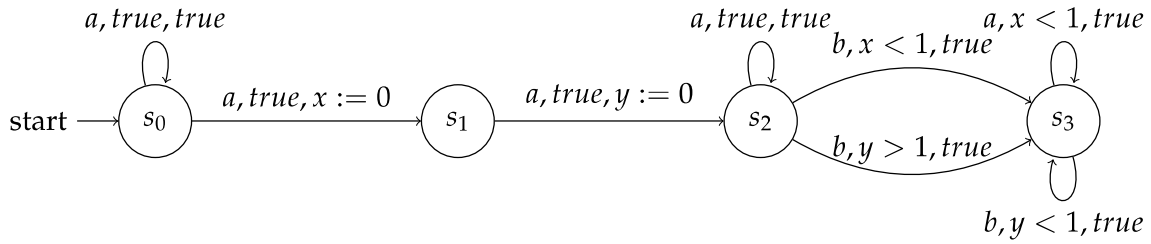


Figure 41.: A Timed Automaton with four states

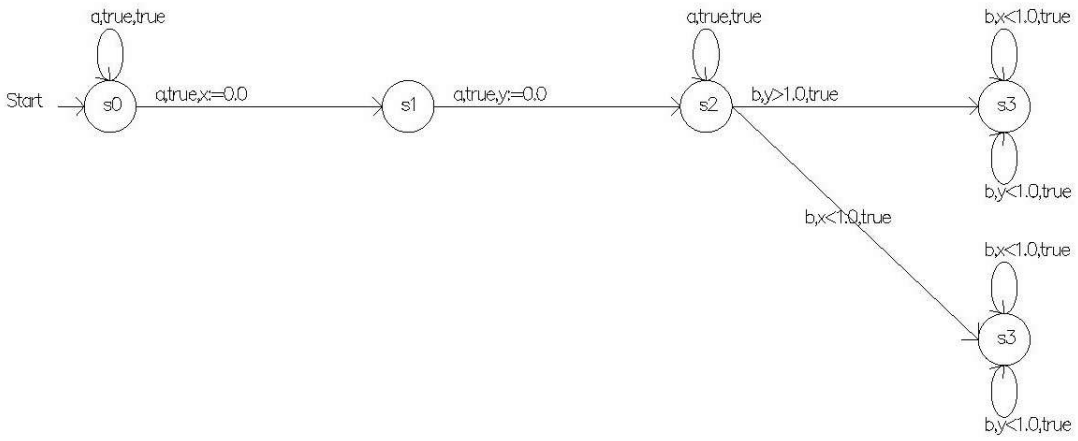


Figure 42.: A timed automaton with four states generated through our algorithm

In the last state, q_4, note that the transitions do not convert into a single state. It occurs because the algorithm acts recursively, then tries to draw the last state, it first checks if the state is already drawn. It is related to the generation of a regular expression, which has two references to the same state and not to the automaton drawing.

$$
\begin{aligned}
&s_0 = \mu_{s_0}((a, true, true).s_0 + (a, true, x := 0).X) \quad \textit{Where} \\
&X = \mu_{s_1}((a, true, y := 0).Y) \\
&Y = \mu_{s_2}((a, true, true).Y + (b, x < 1, true).Z + (b, y < 1, true).Z) \\
&Z = \mu_{s_3}((b, x < 1, true).Z + (b, y > 1, true).X)
\end{aligned}
$$

Figure 43.: A timed regular expression

```
Mu s0 ((a,true,true).s0 + (a,true,x:=0.0).X) Where
X = Mu s1 ((a,true,y:=0.0).Y)
Y = Mu s2 ((a,true,true).Y + (b,x<1.0,true).Z + (b,y>1.0,true).Z)
Z = Mu s3 ((b,x<1.0,true).Z + (b,y<1.0,true).Z)
```

Figure 44.: A timed regular expression generated through our algorithm

Example 4.3.6. *Consider a timed automaton in Fig 45, from the article Alur and Dill (1994). Begins in state s_1, and, it can either bounce around between states s_0 and s_1 indefinitely and, then transits to state state s_2; or, it can transit infinitely between the states s_2 and s_3. Alternatively, it can transit straight away to state s_2 and end up in a set of states which can switch infinitely often the set states s_2,s_3.*

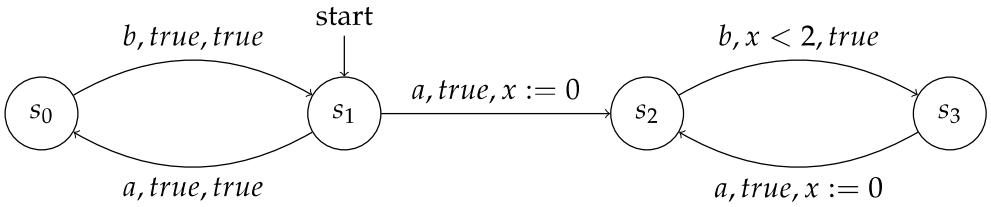


Figure 45.: A timed automaton with four states



Figure 46.: A timed automaton with four state generated through the algorithm

$$
\begin{aligned}
&s_0 = \mu_{s_0}((a, true, true).X) \quad Where \\
&X = \mu_{s_1}((b, true, true).s_0 + (a, true, x := 0).Y)) \\
&Y = \mu_{s_2}((b, x < 2, true).Z) \\
&Z = \mu_{s_3}((b, true, x := 0).Y)
\end{aligned}
$$

Figure 47.: A timed regular expression

```
Mu s0 ((b,true,true).X) Where
X = Mu s1 ((a,true,true).s0 + (a,true,x:=0.0).Y)
Y = Mu s2 ((b,x<2.0,true).Z)
Z = Mu s3 ((a,true,x:=0.0).Y)
```

Figure 48.: A timed regular expression generated through the algorithm

Example 4.3.7. *This last timed automaton is the most complex one until now, from the article Alur and Dill (1994). The automaton starts in state s_0, and, after the next state, s_1, it will perform an unspecified number of transitions between the states - s_1, s_2, s_3.*

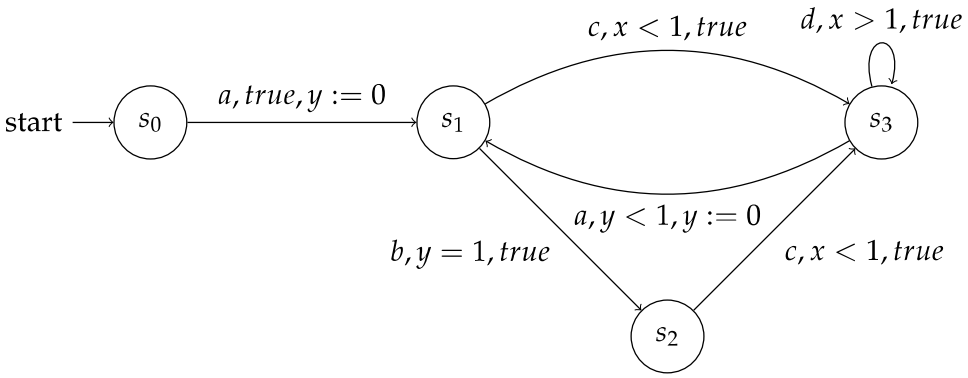


Figure 49.: A timed automaton with four state

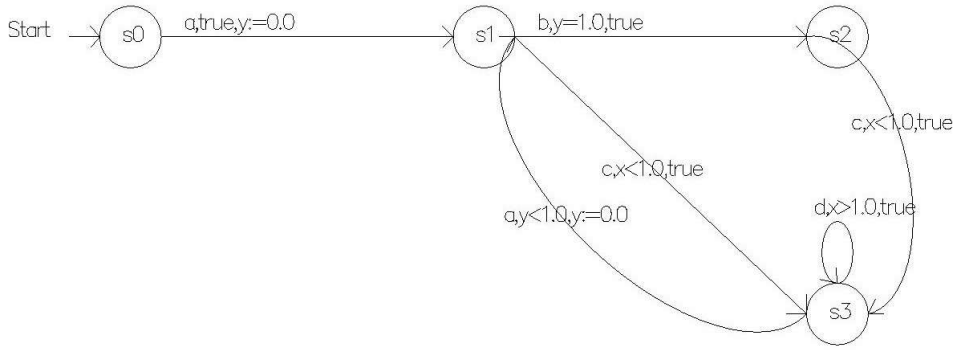


Figure 50.: A timed automaton generated through our algorithm

Despite the automaton generated in Fig.50 having a different arrangement from the automaton in Fig.49, both have the same transitions.

$$
\begin{aligned}
&s0 = \mu_{s_0}(a, true, y := 0).X \quad Where \\
&X = \mu_{s_1}((c, x < 1, true).Z + (b, y = 1, true).Y) \\
&Y = \mu_{s_2}((c, x < 1, true).Z) \\
&Z = \mu_{s_3}((d, x > 1, true).Z + (a, y < 1, y := 0).X)
\end{aligned}
$$

Figure 51.: A timed regular expression

```
Mu s0 ((a,true,y:=0.0).X) Where
X = Mu s1 ((c,x<1.0,true).Y + (b,y=1.0,true).Z)
Y = Mu s3 ((a,y<1.0,y:=0.0).X + (d,x>1.0,true).Y)
Z = Mu s2 ((c,x<1.0,true).Y)
```

Figure 52.: A timed regular expression generated through our algorithm

Sometimes the order of the unfolding expression can be different, like in this example; however, it is not relevant because it is equivalent.

5

IMPLEMENTATION

In this section we will explain in some detail each component of the tool. For the implementation of this algorithm, we had to choose a programming language. Since functional paradigm is closer to the algorithm we want to implement, and within the functional paradigm, the language I'm most familiar with is Haskell, we chose this language.

5.1 OVERVIEW

The code of the implementation is divided into five modules:

1. Parser - Receives a timed automaton, checks input errors and produce an Intermediate structure;
2. Parser Utils - separats the parsing process;
3. Regular Expression - receives a timed automaton (in a processed formed) and generates the corresponding timed regular expression;
4. Draw - Draws a timed automaton from a timed regular expression;
5. Main - Coordinates all the modules previously described and provides a user interface;

5.1.1 *Parser & Parser Utils*

The module Parser receives information of an automaton in a particular format, and converts it into an intermediate structure. The input structure is the following tuple:

1. Initial State - a single state;
2. Set of States - all states of the automaton;
3. Set of Clocks - all clocks of the automaton;
4. Set of Pairs - all clocks associated with a state;
5. Set of Transitions - all transitions;

Our implementation makes sure that the input has the right format via a built-in error detection. By correct format, we mean that elements are introduced in the order mentioned above and separated by a comma. The '\n' is the end of every line. Input tuples have a specific number of elements, e.g. the transitions have three parts (state, extended label, state), separated by a comma.

The names given as an input, clock and state, may only have letters from 'a' to 'z', capital or not, and numbers. The programme will ignore any other symbol or blank space between the two elements. The code below receives an input file and converts it into an intermediate structure.

```
parseFile :: String -> Either ParseError TAut
parseFile = parse fileParser "(unknown)"
  where fileParser =
          do initSt <- nameContent --  Initial State
             eol
             stList <- parserNameList --  List of States
             eol
             clList <- parserNameList --  List of Clocks
             eol
             clAndStList <- (parserList nameContent clockContent)
                            <|> parseNoClock    --  List  (StateName,
Clock) or non
             eol
             transList <- parserTransList --  List of (StateName,
Label,StateName)
```

```
                    return ( (initSt,stList),(clList, clAndStList),
transsList)
```

When an error is detected, a message describing the error pops up. The error information contains the line and column for locating the error. If no error is detected, the program produces the representation of the automaton as an intermediate structure. This structure is used to simplify data manipulation along the process.

```
Error:
"(unknown)" (line 5, column 26):
unexpected "("
expecting " ", "," or "\n"
```

Figure 53.: Error message

A few quick notes. The initial state is expecting just a single state. The set of pairs refers to clocks associated with states, has two options: the state does not have any clock associated, or the state has one or more clocks associated. In the first option is marked by a statement *no*, indicating that there is no clock associated with the state. The second option is to introduce a pair (State, Clock) or a set of pairs {(State, Clock)}.

Example 5.1.1. *The following timed automaton in the format ready to enter the algorithm.*

```
q1
q1,q2,q3
c1
(q1,c1)
(q1,(a,c1:[1,2],true),q2),(q2,(b,true,c1:=0),q3),(q3,(c,c1:[2,3],true),q1
)
```

Here, we see the input introduced in line four to the structure previously mentioned the pair (State, Clock), which indicates that the state q_1 has clock c_1 associated. If there is no pair to introduce, the "no" statement should be placed instead.

5.1.2 *Regular Expressions*

The Regular Expressions module aims at producing timed regular expression. For this, we convert it from the previous intermediate structure into Brzozowski expressions. After that, the expression is built. It also is necessary to verify if all clocks mentioned are valid; by valid, we mean that the code can process the automaton without popping an error message.

Below we have all the structures used to generate timed expressions. Note that we may have states with no outgoing transitions, and by default, this is represented by the expression *ExpTrue*.

```
-- | Atomic predicate of Reset Structure
data AtomReset = AtomReset String String Double |
                 Tru  String
                  deriving (Show,Eq)

-- | Reset Structure
data Reset  = AtmR AtomReset |
              AndR Reset Reset
                  deriving (Show,Eq)

-- | Atomic predicate Structure
data AtomPred = AtomPred String Double Double |
                AtomL0 String String Double   |
                AtomL1 Double String String   |
                AtomL2 String String String   |
                Tr  String
                  deriving (Show,Eq)

-- | Guard Structure
data Invar =  Atom AtomPred   | -- ^ Atomic
              Or Invar Invar  | -- ^ Disjuntion
              And Invar Invar | -- ^ Conjuntion
              Not Invar         -- ^ Negation
                deriving (Show,Eq)

data Expr =  Cl ClockName                   | --  Clock
             St StateName                   | --  State
```

```
Init StateName                    | --  Initial state
Plus Expr Expr                    | --  Plus
Inc String                        | --  Incognito
Miu Expr Expr                     | --  Locker
Trans ((Char,Invar,Reset),Expr)   | --  Transition
Conj (Expr,Expr)                  | --  Conjunction
ExpTrue
   deriving (Show,Eq)
```

The function *updatedtable* takes as an argument a table of Brzozowski Equations, updates the table of expressions, making the necessary substitutions in each of the expressions on every iteration. It ends when n is at least equal to the number of expressions.

```
updatedtable:: Int -> [(StateName,Expr)] ->(Int,[(StateName,Expr)])
updatedtable n l = if n >= length l then (n,l)
            else let (s,t) = getIt (n,l)
                     l'= (auxilary)  (freeVar t,(s,t),l)
                     in updatedtable (n+1) l'
```

Example 5.1.2. *Consider the timed automaton in Fig. 54.*

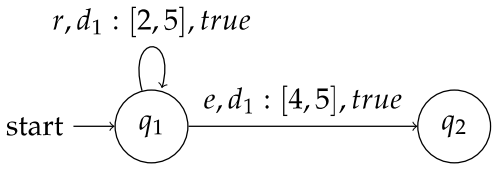


Figure 54.: Timed automaton

The Brzozowski Equations are:

$$q_1 = \mu_{q_1}(r, d_1 : [2,5], true).q_1 + (e, d_1 : [4,5], true).q_2$$
$$q_2 = \mu_{q_2} true$$

We have one substitution locked (q1) in this expression, and the other is free (q2). Therefore only one substitution will occur (q2).

$$q_1 = \mu_{q_1}(r, d_1 : [2,5], true).q_1 + (e, d_1 : [4,5], true).(\mu_{q_2} true)$$
$$q_2 = \mu_{q_2} true$$

All expressions are now locked, so there are no more substitutions.

The presentation of the regular expression is a bit different from what we are used to, but, as the expressions are long, we feel the need to do something to improve. To tackle this issue, we came up with a way of displaying such regular expressions in a simple form. The first expression is presented in full size, and at the end of it, we put the *Where* statement to denote the unfolding of the expression. In the case of a single expression, the *Where* statement is not used. This way, we present all the expressions in several lines, which improves readability and avoids lengthy expressions.

As we used this statement and listed the expressions, we also saw it would be necessary to ”assign” expressions to ”variable”. This ”assignment” is merely illustrative, it is made to facilitate the interpretation of expressions, so there is no true ”assignment”. A variable is denoted by a capital letter.

Example 5.1.3. *Consider the expression mentioned above.*

$$q_1 = \mu_{q_1}(r, d_1 : [2,5], true).q_1 + (e, d_1 : [4,5], true).(\mu_{q_2} true)$$

Assign the ”variable” X with the expression $\mu_{q_2} true$, $X = \mu_{q_2} true$.

$$q_1 = \mu_{q_1}(r, d_1 : [2,5], true).q_1 + (e, d_1 : [4,5], true).X \quad Where$$
$$X = \mu_{q_2} true$$

The following schema summarizes the behavior of this module.

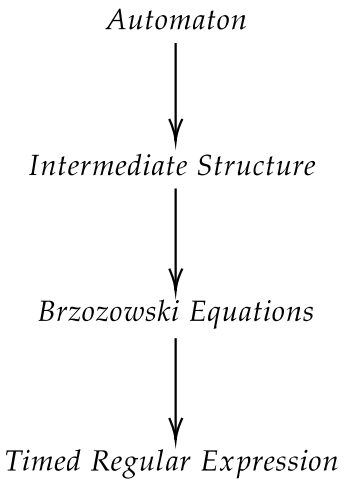


Figure 55.: Diagrammatic description of the regular expressions

5.1.3 *Draw*

The module Draw prints the final expression along with the corresponding automaton. The automaton is constructed from the generated expression. To draw the automaton, we used the Haskell Library Gloss (`https://hackage.haskell.org/package/gloss`), which is easy to work with.

The process of drawing an automaton is divided into the following steps. Firstly, we begin by decomposing the timed regular expression resulting in a set of transitions. This set is divided into two parts: ingoing and outgoing transitions. We made this distinction because the functions that draw them receive different parameters. As states we draw, their positions are saved in a list, preventing repetitions.

When drawing the states, we have to pay attention to the length of their names. It is vital to avoid writing over other elements of the drawing. We want as well that the names of the states are well-framed within the circle. So we generate the a circle according to the size of all existing states. Similarly, the same is done for labels that often increase in length due to more complex guard's introduction. The final result is a timed automaton.

5.1.4 *Main*

The Main module couples every module together and creates a window for file managing. This file manager helps the user to choose the file that he wants to parse. The interface is similar to other file managers.

In case of an error, while parsing a file, it will appear a file called *Error.txt*. This file contains an error message, which indicates the line and column in which it occurs and a suggestion to correct it. It also has a filter for files terminating in ".txt" and other option to see all files.

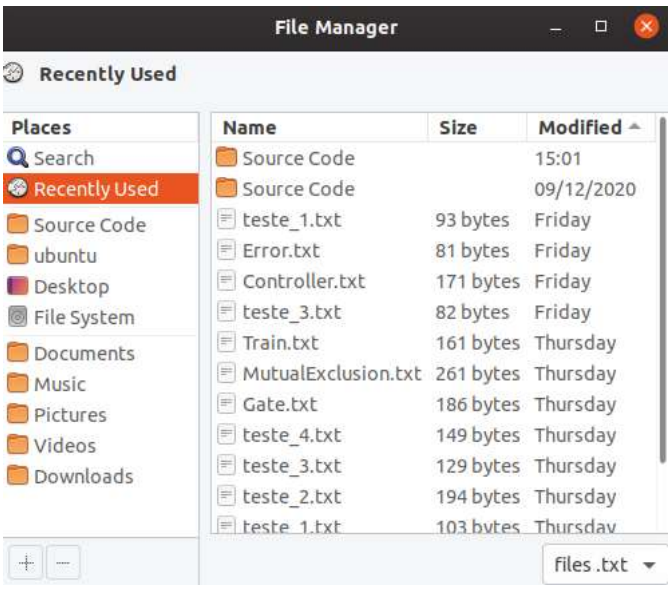


Figure 56.: File manager

5.2 KEY IDEAS IN THE IMPLEMENTATION

Whilst developing the implementation, we encountered some obstacles, which we briefly describe next. The first one concerns to the possibility of non-termination in the generation of regular expressions. We can not let substitution occur and hope that will stop. Indeed, there are some cases where non-termination appears. So there is a possibility that we can not generate an expression. For solving this problem, we have to limit the number of iterations. From Alexandra Silva thesis [Silva (2010)], we saw that to substitute all expressions, in a substitution algorithm, we need to iterate at most the number of expressions formed.

6

CASE-STUDIES

Up to this moment, we have only presented simple cases of automata and expressions. This chapter discusses two case studies, which are more complex and connected to real-world scenarios: Mutual Exclusion and Railways-crossing system.

6.1 MUTUAL EXCLUSION

We begin with the Mutual exclusion protocol, developed by Fischer. This protocol was introduced for managing access to critical memory sections. The access is coordinated to ensure that only one process can be in the critical section at hand. The protocol ensures a register (*id*) to identify processes [Lynch and Shavit (1992)]. The constant k is a time delay that should be larger than the longest time a process may take to perform a step while trying to get access to its critical section [Proença (2019)]. Each process uses a local clock x.

```
repeat
    repeat
        await id =0
        id:=i
        delay(k)
    until id = i
    (critical section)
    id:=0
forever
```

The protocol proceeds as follow. Initially, the value of the shared variable, *id*, is 0. When it observes that *id* is 0, it will write the value of *i* in *id*. After that, it waits for some time *k* (constant *k* should be larger than the longest time that a process may take to perform while trying to get access to its critical section); furthermore, if *id* has the value I, it is safe to enter the critical section [Kelso and Milne (2005)].

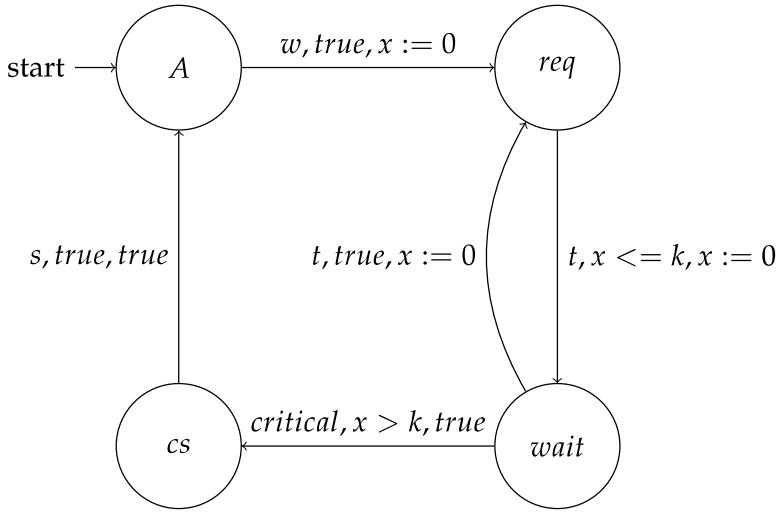


Figure 57.: Mutual Exclusion timed automaton

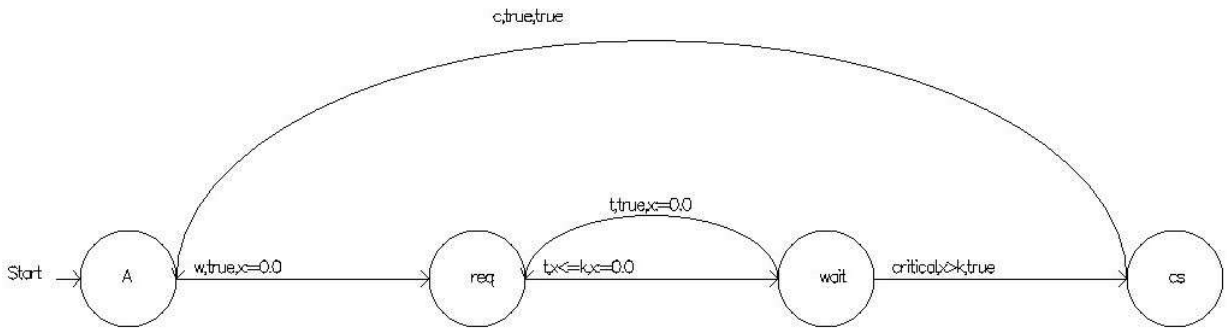


Figure 58.: Mutual Exclusion timed automaton generated

As usual, to build a timed regular expression, we begin from the initial state (A). This state has one transition, labelled by $w, true, x := 0$. This transition constitutes the initial part of the regular expression, $\mu_A(w, true, x := 0).X$ (X represents the next expression); as expected, it is locked. Now, we move to the next state, waiting, which has only one transition, $t, x <= k, x = 0$. This procedure is done for every state. To build a timed automaton, we follow the same procedure. We build a set of transitions for each state. We separate these transitions into the ones that go to another state and remain in the same state.

$$A = \mu_A((w, true, x := 0).X) \quad Where$$
$$X = \mu_{req}((t, x <= k, x := 0).Y)$$
$$Y = \mu_{wait}((t, true, x := 0).X + (critical, x > k, true).Z)$$
$$Z = \mu_{cs}((s, true, true).A)$$

Figure 59.: Mutual Exclusion timed regular expression

```
Miu A ((w,true,x:=0.0).X) Where
X = Miu req ((t,x<=k,x:=0.0).Y)
Y = Miu wait ((t,true,x:=0.0).X + (critical,x>k,true).Z)
Z = Miu cs ((c,true,true).A)
```

Figure 60.: Mutual Exclusion timed regular expression generated

6.2 RAILWAY-CROSSING SYSTEM

Now we present another case study: specifically, a the railway-crossing system. We consider a railroad with several tracks and an unspecified number of trains travelling on the tracks. The example is based on the work of Saeedloei and Gupta (2013). Concretely, we consider three systems: Train, Controller and Gate. The controller ensures the safety and utility properties. The safety property stipulates that the gate must be down while one or more trains are crossing. The utility property states that the gate must be up whenever there is no train in the crossing. All transitions are labelled with events (e.g. lower, up) and with clock restrictions or clock resets [Rehman et al. (2019)]. For instance, the event lower changes the controller state from s1 to s2 if clock constraint $c = 1$ is met.

Train

The automaton that models the train is depicted in Fig.61. The events set is {*approach, idle, out, in, exit*}. The train starts in state t_0. The event *idle* represents that the train is not required to enter the gate. Also, the train communicates with the controller with the events *approach* and *exit*.

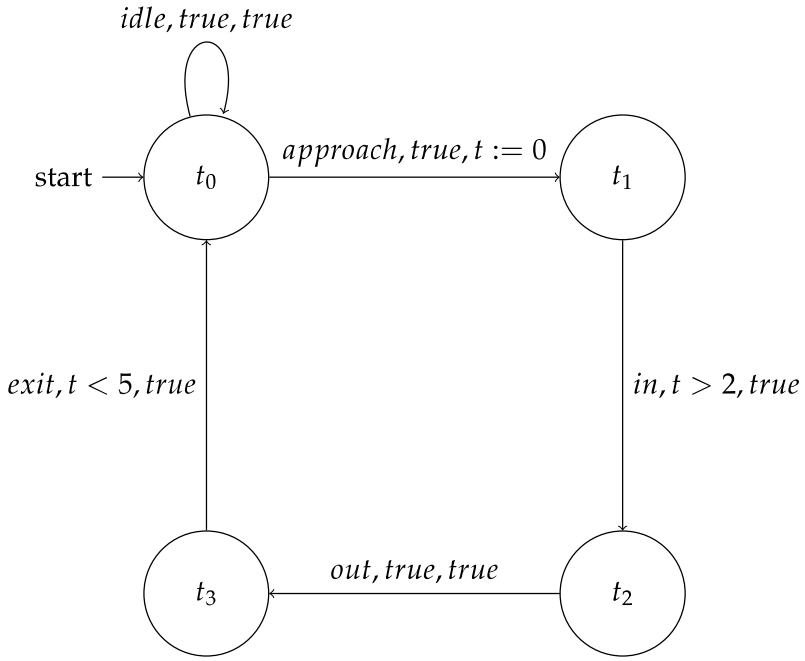


Figure 61.: Train timed automaton

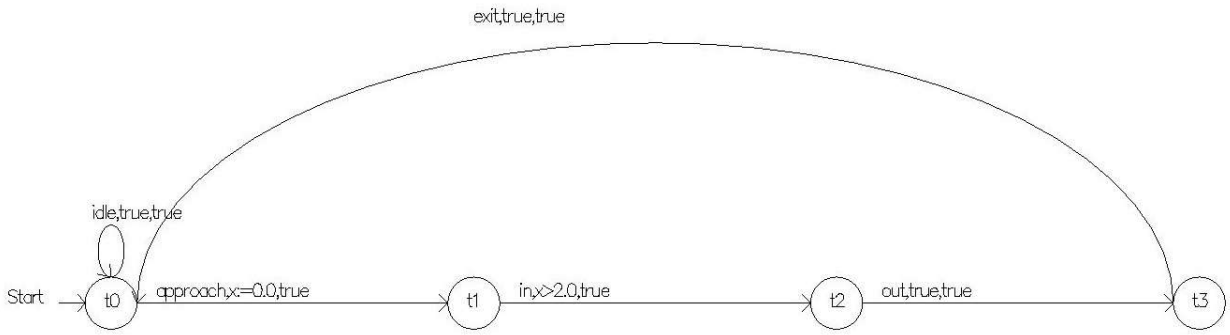


Figure 62.: Train timed automaton generated

The train timed regular expression (presented in Fig. 63) has four sub-expressions. The initial one is $\mu_{t_0}((approach, true, t := 0).X + (idle, true, true).t_0)$. It has two transitions $(approach, true, t := 0).X$, $(idle, true, true).t_0$. As usual, the symbol '+' is used to mark a non-deterministic choice between two possible transitions. In this case, one transition denotes idling (i.e. the only change in the system is time progression), and the other transition informs that the train is approaching the gate. In the next sub-expression, $\mu_{t_1}((in, t > 2, true).Y)$, we have one transition, $(in, t > 2, true).Y$, which informs that the train is passing at the gate. Next, the sub-expression $\mu_{t_2}((out, true, true).Z)$ has one transition, $(out, true, true).Z$, which informs that the train passed the gate but is still close. Finally, the

last sub-expression $\mu_{t_3}((exit, t < 5, true).t_0)$, has one transition, $exit, t < 5, true).t_0$, which informs that the train passed the gate and is relatively far away.

$$
\begin{aligned}
&t_0 = \mu_{t_0}((approach, true, t := 0).X + (idle, true, true).t_0) \quad Where \\
&X = \mu_{t_1}((in, t > 2, true).Y) \\
&Y = \mu_{t_2}((out, true, true).Z) \\
&Z = \mu_{t_3}((exit, t < 5, true).t_0)
\end{aligned}
$$

Figure 63.: Train timed regular expression

```
Mu t0 ((approach,x:=0.0,true).X + (idle,true,true).t0) Where
X = Mu t1 ((in,x>2.0,true).Y)
Y = Mu t2 ((out,true,true).Z)
Z = Mu t3 ((exit,true,true).t0)
```

Figure 64.: Train timed regular expression generated

Gate

The automaton modeling the gate is shown in Fig.65. The event set is { *lower, idle, down, raise, up*}. It communicates with the controller through the events *lower* and *raise*.

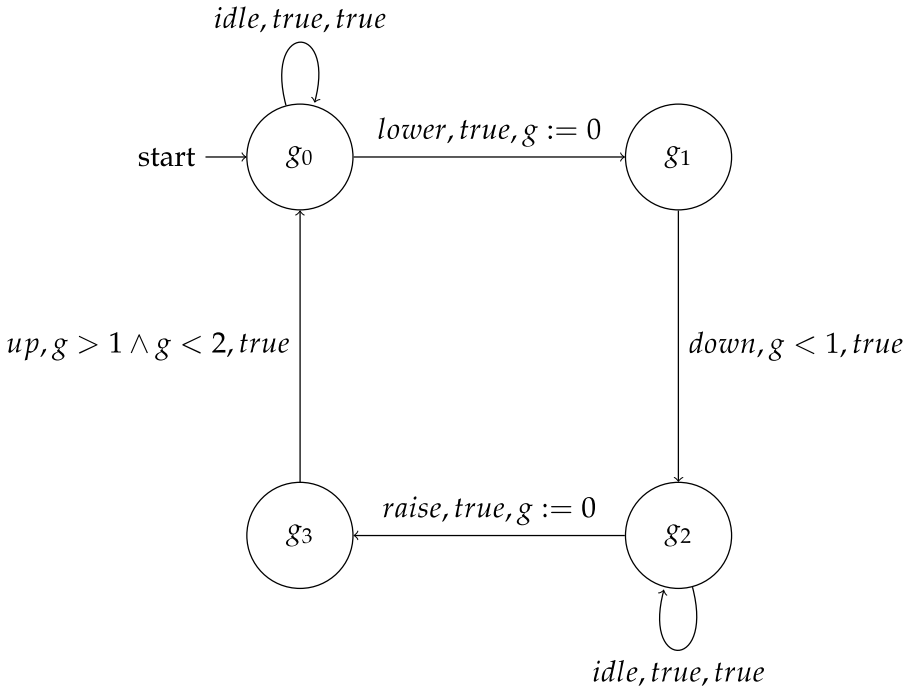


Figure 65.: Gate timed automaton

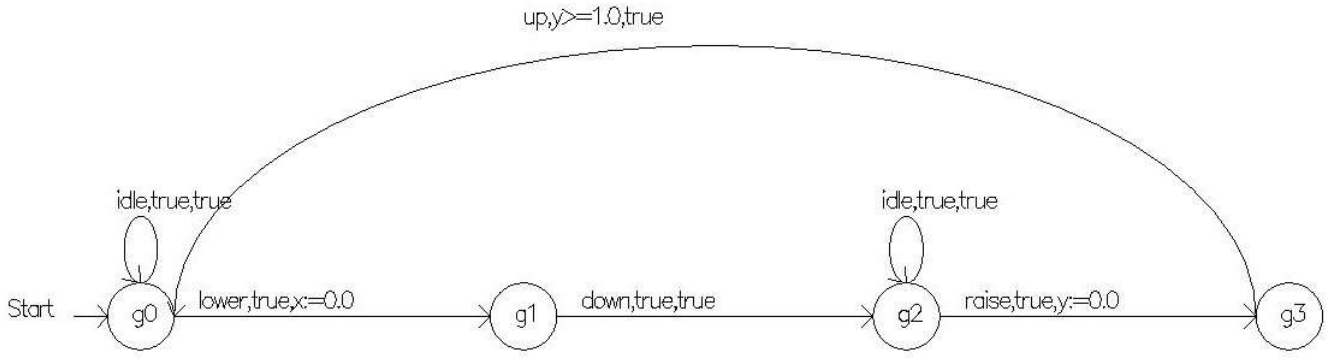


Figure 66.: Gate timed automaton generated

The gate timed regular expression (presented in Fig. 67) has four sub-expressions. The initial one is $\mu_{g_0}((lower, true, g := 0).X + (idle, true, true).g_0)$. It has two transitions $(lower, true, g := 0).X$, $(idle, true, true).g_0$. In this case, one transition informs that the gate is lowering and the other transition denotes idling. In the next sub-expression, $\mu_{t_1}((down, t > 2, true).Y)$, we have one transition, $(down, t > 2, true).Y$, which informs that the gate is fully closed. The other sub-expressions are explained by similar reasoning.

$$
\begin{aligned}
g_0 &= \mu_{g_0}((lower, true, g := 0).X + (idle, true, true).g_0) \quad Where \\
X &= \mu_{g_1}((down, g < 1, true).Y) \\
Y &= \mu_{g_2}((idle, true, true).Y + (raise, true, g := 0).Z) \\
Z &= \mu_{g_3}((up, (g < 2)\&(g > 1), true).g_0)
\end{aligned}
$$

Figure 67.: Gate timed regular expression

```
Mu g0 ((lower,true,x:=0.0).X + (idle,true,true).g0) Where
X = Mu g1 ((down,true,true).Y)
Y = Mu g2 ((raise,true,y:=0.0).Z + (idle,true,true).Y)
Z = Mu g3 ((up,y>=1.0,true).g0)
```

Figure 68.: Gate timed regular expression generated

Controller

Fig.72 shows the automaton modeling the controller. The event set is {*approach,exit,lower,idle*}. When it receives a signal event *approach* from the train, it responds by sending the event *lower* which lowers the gate.

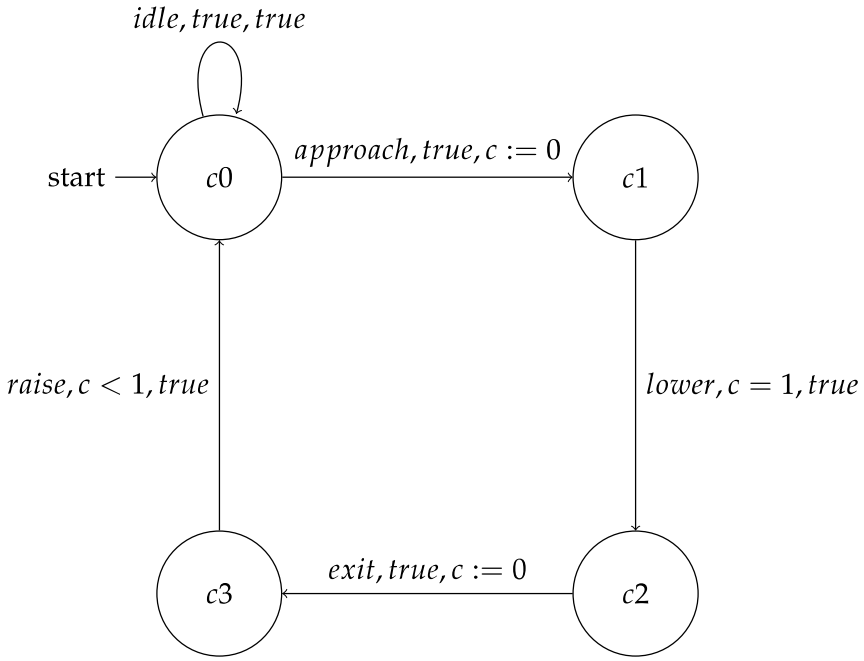


Figure 69.: Controller timed automaton

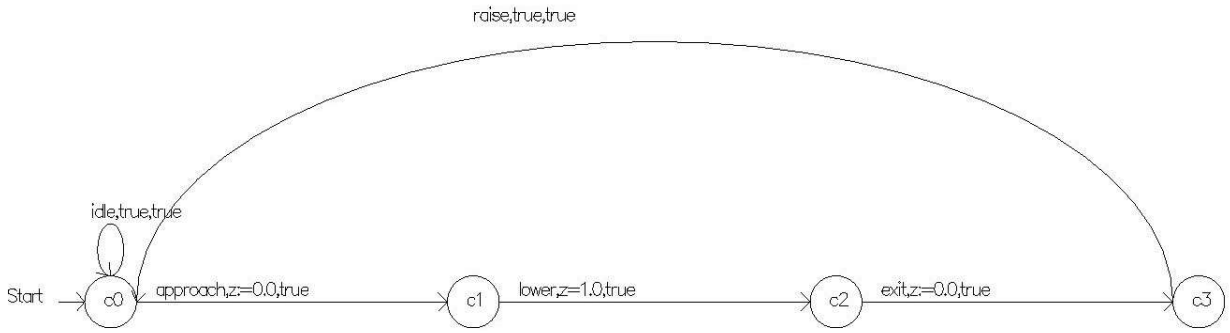


Figure 70.: Controller timed automaton generated

$$c_0 = \mu_{c_0}((approach, true, c := 0).X + (idle, true, true).c_0) \quad Where$$
$$X = \mu_{c_1}((lower, c = 1, true).Y)$$
$$Y = \mu_{c_2}((exit, true, c := 0).Z)$$
$$Z = \mu_{c_3}((raise, c < 1, true).c_0)$$

Figure 71.: Controller timed regular expression

```
Mu c0 ((approach,z:=0.0,true).X + (idle,true,true).c0) Where
X = Mu c1 ((lower,z=1.0,true).Y)
Y = Mu c2 ((exit,z:=0.0,true).Z)
Z = Mu c3 ((raise,true,true).c0)
```

Figure 72.: Controller timed regular expression generated

All the expressions generated correspond to what was expected.

7

CONCLUSION

7.1 SUMMARY

The main goal of this work was to develop a tool to convert a timed automaton into a timed regular expression. We have started by studying the concept of classical automata. Here, though Hopcroft et al. (2007) we understood how beneficial this kind of machines are, their different variants and corresponding definitions. Next, we focused on divergent methods for generating a classical automaton into a regular expression. We saw that the most suitable conversion method to implement in our tool was the Brzozowski method. This method approaches the generation of regular expressions algebraically, and we consider it to be the most intuitive and most manageable one.
Later on, we studied the introduction of temporal notions of automata. Here, we have seen the first attempt to integrate a concept of time in automata. Through the work of Alur and others on their article *A theory of timed automata* [Alur and Dill (1994)] we started to note the differences between the two approaches, classic and timed. It also introduced us to new properties of this model, clock restrictions and a working principle based on executions. These executions are sequences of transitions that produce timed words.
Next, we introduced our definition of a timed regular expression. We presented the rules to produce timed regular expressions, structure, and the algorithm behind all this. We talked about some choices we made and why we did them. These decisions are preceded by some examples, which show both the expected and generated output. Then, we exemplify the process that converts a timed automaton into a timed regular expression. We explain, step by step, what happens to each expression during this substitution process.
After developing an algorithm that converts timed automata into a timed regular expression, we encoded it in Haskell. We begin by introducing what each module does, with a brief description. This specifies that the information is necessary for its proper processing. We presented the suggested layout, execution and possible errors. Then, we present some

case studies. The first one was a timed automata based on Fischer's protocol for mutual exclusion. This protocol manages racing conditions when processes try to acquire access to a critical section. The second one addresses a common problem of a Railway crossing system. It has three automata - Controller, Train, Gate - which coordinates a set of actions necessary for the system's proper functioning.

Note that the conversion from timed automata to timed regular expressions is useful because regular expressions have a notion equivalence, which an automaton can not provide. This point is worth justifying such an effort to develop this program.

7.2 FUTURE WORK

We present the following lines of future work.

1. Dynamic drawing
   At this moment, we present both the expression and automaton in a static way. We could start by giving the regular expression, and since it can have several lines, it could present one line at a time. And after that, we can represent an automaton to draw each state and transitions one after the other.

2. Use a better rendering library
   As we saw, the examples presented in sections 5 and 6, the output resolution of the Gloss Library is not the best one, and we could not increase it. So it would be a good idea to try other Library to improve the output quality and produce images without aliasing.

3. Expand to Hybrid Automata
   The possibility of expanding to hybrid automata, may add more value to this algorithm as this will allows us to deal with more complex automata. The conversion to a regular expression can be helpful for this class of automata.

BIBLIOGRAPHY

Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling. 1: Parsing*. Prentice-Hall, 1972. ISBN 0139145567. URL https://www.worldcat.org/oclc/310805937.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006. ISBN 0321486811.

Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126 (2):183–235, 1994.

Rajeev Alur and P. Madhusudan. Decision problems for timed automata: A survey. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2004. doi: 10.1007/978-3-540-30080-9_1. URL https://doi.org/10.1007/978-3-540-30080-9_1.

Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Workshop on the Theory of Hybrid Systems, Denmark. 1992*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer, 1993.

Eugene Asarin, Paul Caspi, and Oded Maler. A kleene theorem for timed automata. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*, pages 160–171. IEEE Computer Society, 1997. doi: 10.1109/LICS.1997.614944. URL https://doi.org/10.1109/LICS.1997.614944.

Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. 3098: 87–124, 2003. doi: 10.1007/978-3-540-27755-2_3. URL https://doi.org/10.1007/978-3-540-27755-2_3.

Gérard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theor. Comput. Sci.*, 48(3):117–126, 1986. doi: 10.1016/0304-3975(86)90088-5. URL https://doi.org/10.1016/0304-3975(86)90088-5.

Amnon H. Eden. *B. Jack Copeland (ed), The Essential Turing - Oxford University Press, New York, 2004, ISBN 0-19-825080-0, 613, £29.95*, volume 17. 2007. doi: 10.1007/s11023-007-9051-9. URL https://doi.org/10.1007/s11023-007-9051-9.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007. ISBN 978-0-321-47617-3.

Joel Kelso and George J. Milne. Properties as processes: Their specification and verification. In Farn Wang, editor, *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, volume 3731 of *Lecture Notes in Computer Science*, pages 503–517. Springer, 2005. doi: 10.1007/11562436_36. URL https://doi.org/10.1007/11562436_36.

S. C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies. (AM-34)*. Princeton University Press, 1956. doi: 10.1515/9781400882618-002. URL https://doi.org/10.1515/9781400882618-002.

Dexter Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Inf. Comput.*, 110(2):366–390, 1994. doi: 10.1006/inco.1994.1037. URL https://doi.org/10.1006/inco.1994.1037.

Nancy A. Lynch and Nir Shavit. Timing-based mutual exclusion. In *Proceedings of the Real-Time Systems Symposium - 1992, Phoenix, Arizona, USA, December 1992*, pages 2–11. IEEE Computer Society, 1992. doi: 10.1109/REAL.1992.242681. URL https://doi.org/10.1109/REAL.1992.242681.

Christoph Neumann. Converting deterministic finite automata to regular expressions. *URL http://neumannhaus. com/christoph/papers/2005-03-16. DFA to RegEx. pdf*, 2005.

Renato Neves and Luís Soares Barbosa. Hybrid automata as coalgebras. In Augusto Sampaio and Farn Wang, editors, *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings*, volume 9965 of *Lecture Notes in Computer Science*, pages 385–402, 2016. doi: 10.1007/978-3-319-46750-4_22. URL https://doi.org/10.1007/978-3-319-46750-4_22.

José Proença. Time-critical reactive systems (modelling), 2019. URL https://alfa.di.uminho.pt/~nevrenato/pdfs/timedautomata.pdf.

Michael O. Rabin and Dana S. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, 1959. doi: 10.1147/rd.32.0114. URL https://doi.org/10.1147/rd.32.0114.

Aniqa Rehman, Saba Latif, and Nazir Ahmad Zafar. Automata based railway gate control system at level crossing. In *2019 International Conference on Communication Technologies (ComTech)*, pages 30–35, 2019. doi: 10.1109/COMTECH.2019.8737833.

Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249 (1):3–80, 2000. doi: 10.1016/S0304-3975(00)00056-6. URL https://doi.org/10.1016/S0304-3975(00)00056-6.

Neda Saeedloei and Gopal Gupta. Timed π-calculus. In Martín Abadi and Alberto Lluch-Lafuente, editors, *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, volume 8358 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013. doi: 10.1007/978-3-319-05119-2_8. URL https://doi.org/10.1007/978-3-319-05119-2_8.

Shmuel Safra. On the complexity of omega-automata. In *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, pages 319–327. IEEE Computer Society, 1988. doi: 10.1109/SFCS.1988.21948. URL https://doi.org/10.1109/SFCS.1988.21948.

Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13 (1):158–169, 1966. doi: 10.1145/321312.321326. URL https://doi.org/10.1145/321312.321326.

Alexandra Silva. *Kleene Coalgebra*. PhD thesis, Radboud Universiteit Nijmegen, 2010.

A

INSTRUCTION MANUAL FOR INSTALLATION AND USE

There are two options to use the program. Install the necessary programs as well as their dependencies in your pc; or use a virtual machine already with all programs installed in a Linux environment. First, let's detail the necessary instructions to install the programs properly.

1. Install Haskell Platform from the website (https://www.haskell.org/downloads/);
2. Install cabal with following command *sudo apt-get cabal install*;
3. Install the library libgtk with the command line *sudo apt install libgtk2.0-dev*;
4. Install the dependencies using command, *cabal install*, followed by the dependency name;
   Dependencies name:
   - Gloss;
   - Ghc;
   - Stack;
   - Parsec-3.1.13.0;
   - Parsec-numbers-0.1.0;
   - Split;
   - Gtk2hs-buildtools;
   - Gtk;

5. Make an executable using the following command, *ghc -o Regular Main.hs*;

Note that you should perform the command, *ghc -o Regular Main.hs*, in the folder with all the modules files. After executing this, new files will be created, including the executable file named *Regular*.
The other option is to use the virtual machine with the lubuntu operating system from https://www.osboxes.org/virtualbox-images/. To use the algorithm, you must find the executable, which is in the folder called *Codigo fonte* and then double click on the file Regular. In that folder it has another folder with the case studies named, *Casos de estudo*, and the code module files. The password to access the virtual machine is *osboxes.org*.

B

EXAMPLES

This appendix shows the content of files given as inputs to the program, which, generate the examples presented in sections 4.3 and 6.

B.1 LEVEL 1

In this first level, we present the most simple examples contained in section 4.3.

B.1.1 *Example 1*

```
q1
q1,q2,q3
c1
(q,c1)
(q1,(a,c1:[1,2],true),q2),(q2,(b,true,c1:=0),q3),(q3,(c,c1:[2,3],true),q1
)
```

B.1.2 *Example 2*

```
s0
```

```
s0
x
no
(s0,(a,x<=3,true),s0),(s0,(b,x=3,x:=0),s0),(s0,(c,x<3,true),s0)
```

B.1.3 *Example 3*

```
s0
s0,s1,s2
x
(s0,x)
(s0,(a,x<2,true),s1),(s1,(b,true,x:=0),s0),(s0,(c,true,true),s2)
```

B.2 LEVEL 2

This level contains examples more complex than level 1. And also contained in section 4.3.

B.2.1 *Example 1*

```
s0
s0,s1,s2
x
no
(s0,(a,x<5,true),s1),(s1,(b,true,x:=0),s0),(s0,(a,x<2,true),s2),(s2,(c,
true,x:=0),s0)
```

B.2.2 *Example 2*

```
s1
s1,s2,s3,s4
x,y
no
(s1,(a,true,true),s1),(s1,(a,true,x:=0),s2),(s2,(a,true,y:=0),s3),
(s3,(a,true,true),s3),(s3,(b,x<1,true),s4),(s3,(b,y>1,true),s4),(s4,(b,x
<1,true),s4),(s4,(b,y<1,true),s4)
```

B.2.3 *Example 3*

```
s0
s0,s1,s2,s3
x
no
(s0,(a,true,true),s1),(s1,(b,true,true),s0),(s0,(a,true,x:=0),s2),(s2,(b,
x<2,true),s3),(s3,(a,true,x:=0),s2)
```

B.2.4 *Example 4*

```
s0
s0,s1,s2,s3
x,y
no
(s0,(a,true,y:=0),s1),(s1,(c,x<1,true),s3),(s3,(a,y<1,y:=0),s1),(s2,(d,x
>1,true),s3),(s1,(b,y=1,true),s2),(s2,(c,x<1,true),s3)
```

B.3 LEVEL 3

This level contains the inputs of the case studies - Mutual Exclusion and Railway-Crossing system -both contained in section 6.

B.3.1 *Fischer protocol*

```
sleeping
sleeping,waiting,trying,critical
lock
no
(sleeping,(w,lock=0,true),waiting),(sleeping,(s,lock!=0,true),sleeping),(
    waiting,(t,lock:=i,true),trying),(trying,(s,lock=i,true),sleeping),(
    trying,(c,lock=i,true),critical),(critical,(s,true,lock:=0),sleeping)
```

B.3.2 *Controller*

```
raise
raise,approach,in,exit
c
no
(raise,(a,c:=0,true),approach),(approach,(i,c=1,true),in),(in,(e,c:=0,true),
    exit),(exit,(r,c<1,true),raise),(raise,(r,true,true),raise)
```

B.3.3 *Gate*

```
up
up,lower,raise,down
g
no
(up,(l,g:=0,true),lower),(lower,(d,g<1,true),down),(down,(d,true,true),down)
    ,(down,(r,g:=0,true),raise),(raise,(u,(g<2)&(g>1),true),up),(up,(u,true,
    true),up)
```

B.3.4 *Train*

```
exit
exit,approach,in,out
t
no
(exit,(a,true,t:=0),approach),(approach,(i,t>2,true),in),(in,(o,true,true),
    out),(out,(e,t<5,true),exit),(exit,(e,true,true),exit)
```

NB: place here information about funding, FCT project, etc., in which the work is framed. Leave empty otherwise.